Table of Contents

FOREWORD

This book is the product of me (Jim) trying something new in our introductory Computer Science course at the University of Manitoba. Through the process of trying to see how I can teach programming using Processing, I started to panic as none of my examples worked, and ended up writing copious notes to help myself work through the problems. The first draft of those notes were used in the first term we taught programming using Processing. Following, there has been a major overhaul for content and exercises, to the version you see now. Finally, I am ready to toss this online, in hopes it may be useful to others.

Acknowledgments

I really need to thank the University of Manitoba Computer Science Department for this book. While I wrote the text here, many of the strategies, techniques, and explanation methods are drawn from the ethos kicking around our department on how we teach programming. This is as much my own teaching style as what I have learned from all the great teachers around here. In particular, I need to thank Andrea Bunt, who was brave enough to try a Processing assignment with me, John Anderson who gave the red light to do a whole course this way, and John Bate, who helped immensely with the material and many of the exercises: in fact, some are straight-up taken from John's labs in prior terms.

History

2016-may-31 – Version 1.0 live!

License

The following text is taken from the above link:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial — You may not use the material for commercial purposes.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

(page intentionally left blank)

UNIT 1. INTRODUCTION

Each unit will start with the three sections below: **Summary**, **Learning Objectives**, and **How to Proceed**. The **Summary** will just provide an overview of the activities and topics covered, and the **How to Proceed** will be a simple description of how to approach the material. The **Learning Objectives** are particularly important as they describe the skills you will have by the end; once finishing the unit, return to this section to be sure that you feel comfortable with all the items listed.

1.1 Text Book Annotations and Aids

This course has the following aids to help you out:

🎯	[1]Learning Objectives at the beginning of each Unit. Check yourself against these to be sure you are learning the required skills.
❗	[2]Important sections that you should pay special attention to.
🎓	[3]Academic sections for interest only, which are not testable.
✓ **Check your Understanding**	[4]Sections with exercises to check your understanding. These sections are your primary way to develop your skills and practice – work hard at them.
✓ **How did you do?**	At the end of a unit, go back and review the learning objectives

[1] https://commons.wikimedia.org/wiki/File:Target.svg
[2] https://commons.wikimedia.org/wiki/File:Important-1.svg
[3] https://commons.wikimedia.org/wiki/File:Mortarboard.svg
[4] https://commons.wikimedia.org/wiki/File:Red_check.svg

Summary

In this unit you will assess whether or not this course is for you, and learn briefly about the many things that you can do with programming.

Learning Objectives

After finishing this unit, you will be able to…

- Explain to a layperson what computer programming is
- Calculate how many operations a computer can perform each second based on its specifications

How to Proceed

- Read the unit content.
- Have a web browser open so that you can do the searches recommended.
- Do the sets of exercises in the **Check your Understanding** section.
- Re-check the **Learning Objectives** once done.

1.2 **Introduction**

Programming is awesome! That's all you need to know. Seriously, though, why bother with learning programming? Look around you, computers are everywhere. The obvious ones are your laptop, tablet, and phone – and game system if you're into that. But don't forget about your car (if it's newer than 1980), and even your home appliances (if they're newer than mine, at least). Learning computer programming gives you some insight into how and why these things work. And these days, there are great new tools (toys!) that let you build and test things yourself. Do an internet search for Arduino, Raspberry Pi, or Phidgets, and you'll find large communities of home-cooked cool gadgets by do-it-yourselfers with limited programming background. Don't forget immersive VR, which is emerging as the next cool thing – these are computers, too! And you can use programing to be creative and make your own software with these. So, with basic programming, you'll have the foundation to start hacking away at your own computer programs (games! business tools! art projects!), or even get into hobbyist hardware hacking and inventing. Who knows, maybe you'll go into Computer Science and keep learning more.

1.3 **Do I belong in this course?**

Probably (we need to get our numbers up!). More practically speaking, this is a true introduction course. No programming background or experience is required or expected. You may see people around you who have taken prior courses, but they are ahead of the curve and the course is not aimed at them. So what background do you need? Just basic computer skills – typing, using new software, using the web, etc. Installing new software is helpful, too.

If you do have extensive programming experience, you can still learn a great deal in this course. Many prior students have commented on the additional depth they have learned and dusty corners cleaned out through this (and, ahem, the easy A+?) – You may be surprised at what you will learn. However, there are ways of challenging the course and skipping ahead. Talk to your undergraduate advisor about this.

1.4 **What is Programming?**

So what is this course about? It is about problem solving, it is about solving puzzles with a given set of tools and constraints. Computers are really capable, but they are also really stupid and can't do much on their own. You need to learn specific tools to wrangle the computer into doing what you want it to do. This ends up being a nice set of puzzles, and as such, many people who enjoy puzzles also enjoy programming. That's right, I said *enjoy* – programming can be so enjoyable, in fact, that there is an immense community called the "open source community" where people, in their spare time, create computer software for fun and give it away for free. So, the course is basically a) learn tools that computers understand b) practice by solving puzzles to make the computer do work for you.

Imagine that you are a woodworker – wood is a very versatile and capable material, but you need to learn the techniques and tools (saws, hammers, screws, nails, etc.) to make that wood into a nice new deck. Programming is similar – the possibilities are endless, but you need to master the tools first, and figure out how to use them to make your dreams reality! In this course you will learn these basic tools that will stick with you no matter where you encounter computer programming or in what language.

You will learn abstract thinking and problem solving with respect to how computers work. This will not be a course in learning how to use software like MS Word or Excel, you will not learn how to make webpages, or make a blog, or how to use the internet. We will do cool stuff, however, and you will be more comfortable with understanding how computers work.

Earlier I said that computers are really stupid. So why do we use them? The reason is simple: they are really (really) fast. A typical computer these days can do about 2 Gigahertz and has 4 to 8 cores – translation:

- 2 Gigahertz means 2,000,000,000 operations per second
- 8 cores means it can do 8 calculations in parallel
    - 16,000,000,000 operations per second. (16 billion)

That's a really big number that is basically unfathomable to us mere mortals. But! If we can harness that power, we can put it to work, and we can do a lot of work with 16 billion operations (like watch cute kitten videos on the internet).

For Information Only (not testable): Throughout this course, there will occasionally be information given that is purely for academic purposes. Those interested can sit and think about these items, but they are beyond the scope of the course and are not testable. These sections are marked with a graduation hat to the left. For example, the above Gigahertz example assumes that the computer can do one complete operation every time the clock ticks. This is not true – many operations take multiple ticks. Further, when you have multi-core machines (the above example has 8 cores), it is not as simple as the 8 cores working in parallel. It actually takes work, overhead, to coordinate all the cores and enable them to share resources such as memory or hard disks. It turns out that a great deal of factors determine how much work your computer can get done, not just the clock speed or cores.

Coming back to computers being stupid – if you tell your computer to smack its head against the wall forever, it will do it, just as you tell it to (and probably do it 16 billion times each second). As a programmer, your job is to give clear instructions to the computer so that it does exactly what you want it to. Unfortunately, if there is even one mistake in your programming, the computer will follow that mistake blindly and your program will not work like you expect it to. It has no common sense, and cannot fill in the blanks like a person can.

computers do exactly what you tell them to do, not what you want them to do.

Notice the green exclamation mark to the left? Throughout this course, you will see this mark to signify that there is something really important here. A definition, core concept, or requirement for assignments.

This highlighted statement may seem a little strange right now, but by the time you finish your first assignment, you will clearly understand what this means. As a programmer, your job is to use the tools of programming to give clear instructions to a computer. You need to translate what you want the computer to do into the computer's language.

Given a job to do, like drawing a diagram, you provide the recipe for the computer to follow to do the work. This is kind of like cooking – a perfect recipe will enable you to reproduce a favorite dish or dessert. Similarly, sheet music is a program for a musician to reproduce a piece of music. Unfortunately, in both these cases you give some credit to the cook or musician – with computers, you need to spell out everything and leave nothing to common sense, style, or interpretation.

1.5 **Computer Language**

Different computers speak different languages. Also, the real on-the-chip language that computers need is really confusing and hard to understand – even for computer experts. Basically it is a bunch of 1s and 0s (called *binary*), or a short-hand called *assembly language*, which makes you micro-manage every single computer operation. Assembly is also different across computers, making it hard to remember. These are called *low-level* languages because they are very close to how the machine works.

I know assembly!

```
MOV AL, $FOOD
CMP AL, $BONE
JE .CHEW
```

Easy, right!?

Higher-level languages try to make things more accessible to people by moving further away from the hardware requirements (good news for us puny humans). Some common high-level languages that you may have heard of are C, C++, C#, Java, Python, PHP, etc. In this class, we will be using a language called Processing, which is effectively Java with all the crap removed (err, I mean, simplified for learning?). The other cool thing about Processing is that we will be doing graphics from the beginning!

There are a lot of similarities across computer languages. Processing (and Java) are "C-Like" languages, in that they resemble the classic programming language "C". Many modern languages do. In fact, you will find that most languages are so similar that, once you finish this course, you can pick up many new languages just by reading a quick primer. The main computer language tools you will learn here exist in very similar forms in almost all languages. As such, you are not really learning the Processing language, but rather, learning fundamental computer tools that exist in the vast majority of programming languages.

✓ Check your Understanding

1.6 Check your Understanding: Exercises

In this course, exercises will be posted at the end of the unit. These exercises will not be marked or assessed in any way, but are primarily for you to think about the things in the unit, and to provide structure for you to work on what you learned. Of course, feel free to email your instructor regarding any of the exercises. These exercises do not have solutions posted for learning purposes. If you need help, engage the forums, work in teams, or ask your instructor.

The exercises in this section are primarily about looking around the internet to learn a little and consider what programming really is all about. In later chapters the exercises become much more complex and can be quite difficult.

Exercises in this course are marked as having a difficulty level: bronze, silver, or gold. You should use these to gauge your success in the course and expected grade. For example, if you want to pass the course you need to be able to do all the bronze exercises without difficulty. If you want a B in the course, you need to be finishing the silver exercises. If you want an A or A+, then doing many of the gold exercises is required. If you are struggling with the silver exercises, then be prepared that a B may be difficult to attain.[5]

CuSn
Ag
Au

CuSn Exercise 1. Go to your favorite computer store website, and take a quick look at the computers for sale. Find a machine that you may buy (laptop or desktop, your price range).

a. How fast is the machine in GHz (gigahertz)? You may have to click on a “details” or “specs” link, but the information will be available.
b. How many “cores” does the machine have? That is, how many operations can it do in parallel? This is often specified as “dual” or “quad” core. This may be harder to find.
c. Assuming that the computer can do one whole calculation every time the clock ticks, calculate how many operations that computer can do

[5] The medal images are public domain: https://commons.wikimedia.org/wiki/File:Plakette-bronzefarbenmitZackenamRand.png, https://commons.wikimedia.org/wiki/File:Plakette-goldfarbenmitZackenamRand.png, https://commons.wikimedia.org/wiki/File:Plakette-silberfarbenmitZackenamRand.png

per second, which is the GHz (billion cycles per second) multiplied by the cores.

d. If the computer could look at one person every operation, how many people could it look at in one second? How many people are there in the world?
e. If the computer could look at one star per operation, how long would it take to look at all the stars in the milky way?

Exercise 2. A lot of people are interested in learning to program. Do an internet search for learning how to program and take a look at some of the resources. If there is so much available, why take a course? The answer is usually multi-faceted: a) on-line tutorials can be poorly written and, as such, overwhelming. b) you may need a lot of specialized software and it can be hard to get started. c) on-line tutorials often teach how to do something specific (which you may not want to do), while a course teaches fundamentals that set you up for further learning. d) other reasons, even for those with experience?

Exercise 3. Why bother learning to program? Do a search for this: why learn to program? and look at some of the resources. There are many news articles, essays, and opinion pieces by experienced people that talk about some of the reasons why learning to program can be important for anyone.

a. Assuming you do not plan to be a computer science professional, list three reasons that learning to program may be useful for you.

Exercise 4. Search on the internet for the “open source” movement.

a. Why do people create software for free?
b. Find an example of an open source “operating system” that can be used instead of Windows or MacOS
c. What is the world’s most widely used web server software. Is it open source?

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 2. PROCESSING AND PROGRAMMING BASICS

Summary

In this section, you will…

- Get and install the Processing program which enables you to write Processing computer programs
- Familiarize yourself with the Processing program, and the processing canvas
- Learn basic “syntax”, the rules of writing programs
- Write your very first program, and, encounter your first errors
- Learn about drawing and painting order, and how to change the color of what is drawn

Learning Objectives

After finishing this unit, you will be able to write basic Processing programs that draw simple shapes on a drawing canvas. Specifically, you will be able to:

- Choose and set the canvas size for your programs.
- Set the location where shapes are drawn on the canvas.
- Set the background, outline, and fill colors of your canvas and shapes.
- Draw lines, circles and ellipses, triangles, and points.
- Insert in-line and block comments to describe your program.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

2.1 **Introduction**

Processing (processing.org) is a Java-derivative language designed to enable programmers to quickly and easily learn how to do really cool stuff, mostly through computer graphics. While it is often seen as a teaching language, professionals use it as well for small projects and prototyping. Processing is Java under the hood, so it can grow with you as you become an expert. It also has full 3D graphics built in so you can build some really cutting-edge graphical programs with it.

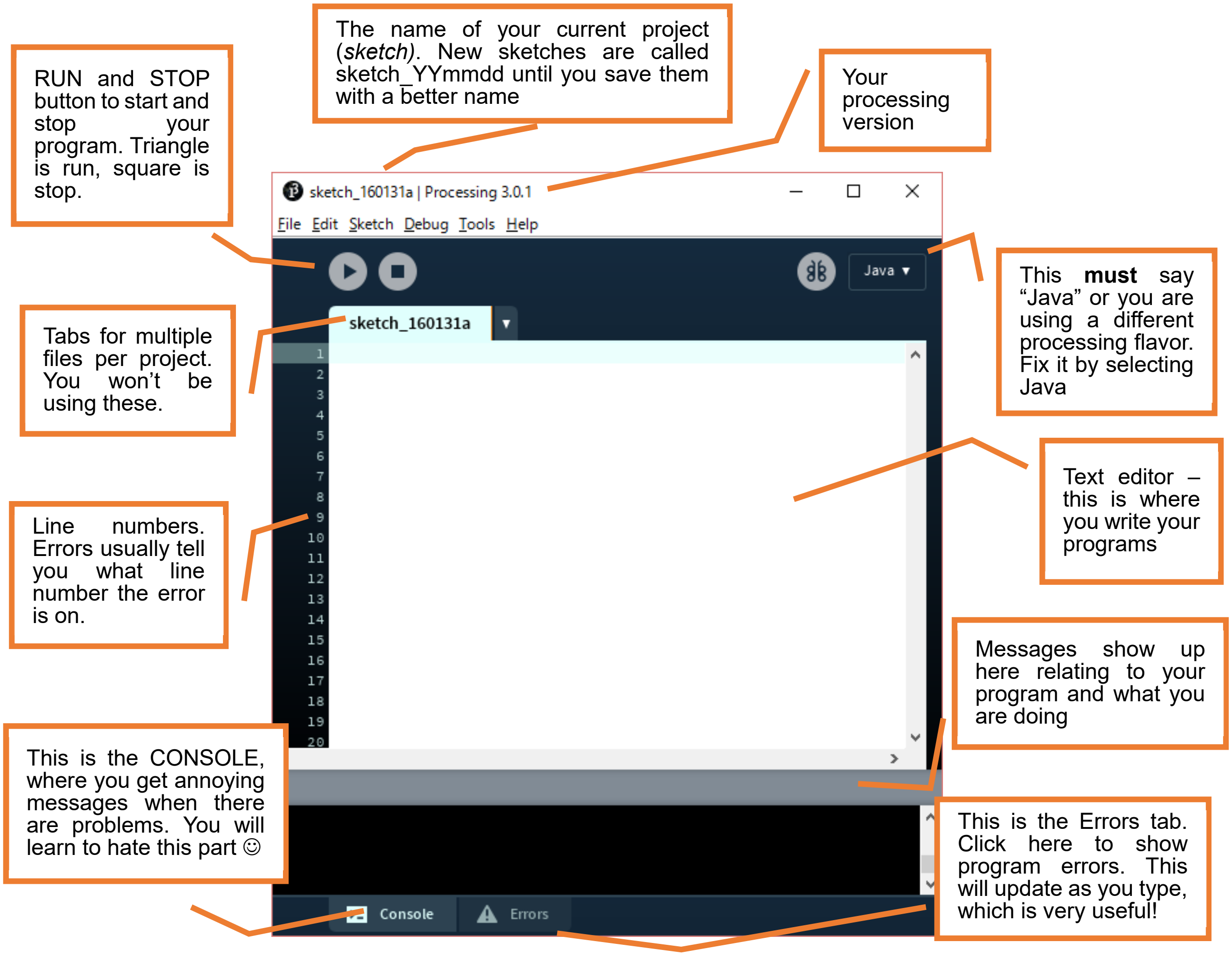
Processing has less testing with Window 8, and you should probably download the 32 bit version, which has been reported to be more stable.

You need to download Processing onto your own computer. Processing is free, and easy to install (ask the help center or the instructor if you're stuck). To start, go to http://processing.org/download and grab the latest version 3. The older versions will work but there will be slight differences in how it looks, so don't bother with that. Also, there is an installation guide for Windows, Mac, or Linux, which can help you to get it running, at https://processing.org/tutorials/gettingstarted/. Once you have it installed, you just double click on the `processing.exe` file and it will start to run.

Once started, you will see the Processing Development Environment. This manages all the under-the-hood parts of computer programming – we'll talk a little about that later in the course. For now, this enables you to learn to write computer programs without worrying about the nitty-gritty details of how the computer goes from what you type, to a program that runs and starts up. This is a good thing.

Processing does a lot of advanced things under the hood. It converts your program to JAVA, then converts that into a language called Java Byte Code, runs a Java Virtual Machine program, and runs your program with that.

Let’s take a closer look at the processing development environment.



sketch_160131a | Processing 3.0.1
File Edit Sketch Debug Tools Help
Java ▾
sketch_160131a
Console
Errors

The name of your current project (*sketch*). New sketches are called sketch_YYmmdd until you save them with a better name

Your processing version

RUN and STOP button to start and stop your program. Triangle is run, square is stop.

Tabs for multiple files per project. You won’t be using these.

Line numbers. Errors usually tell you what line number the error is on.

This is the CONSOLE, where you get annoying messages when there are problems. You will learn to hate this part ☺

This **must** say “Java” or you are using a different processing flavor. Fix it by selecting Java

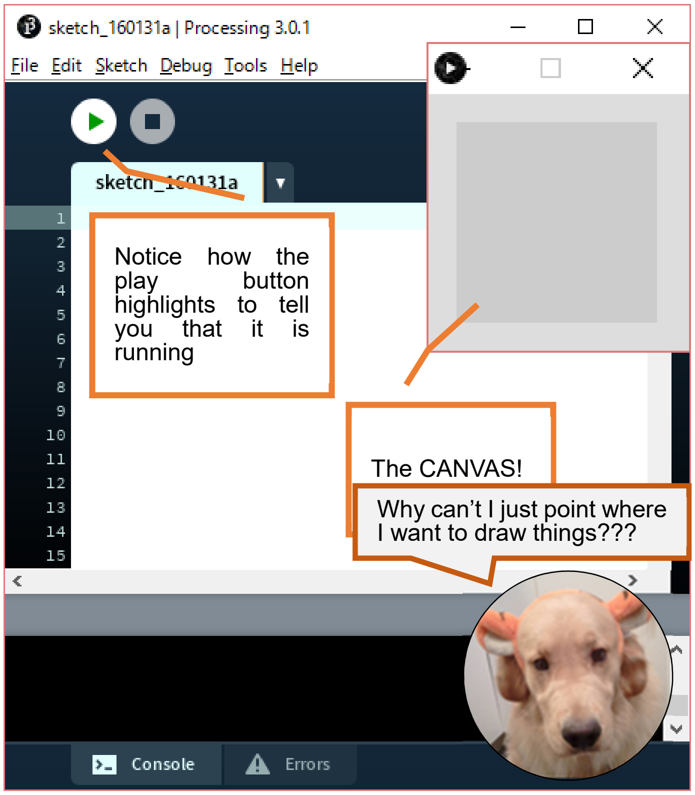
Text editor – this is where you write your programs

Messages show up here relating to your program and what you are doing

This is the Errors tab. Click here to show program errors. This will update as you type, which is very useful!

As the default program name suggests, Processing calls their projects *sketches*. This is fitting, as you get to quickly sketch up your ideas roughly to see how they work. Processing stores your sketches in a *sketchbook*, which is just a folder on your computer.

You can change where these are stored (e.g., ***to a backed up location!!!***) by looking at the `File→Preferences` menu dialog. All your sketches in your sketchbook can also be seen by going to `File->Sketchbook`. Discussing backup options is beyond the scope of this course, but it is highly recommended to have all of your work automatically backed up using one of the many free services currently available. Your University has a plan for a large amount of free storage. In addition, popular options include Dropbox, Microsoft One Drive, or Apple iCloud. Your instructor is unlikely to have patience for problems relating to a crashed hard drive or lost laptop. ***Important:*** Processing saves new sketches in a temporary location until you specifically save it to your own location, by clicking `File->Save`. Be sure to do this immediately when starting a new project so that you do not lose your work if your computer crashes.


sketch_160131a | Processing 3.0.1
File Edit Sketch Debug Tools Help
sketch_160131a
Console
Errors

Notice how the play button highlights to tell you that it is running

The CANVAS!

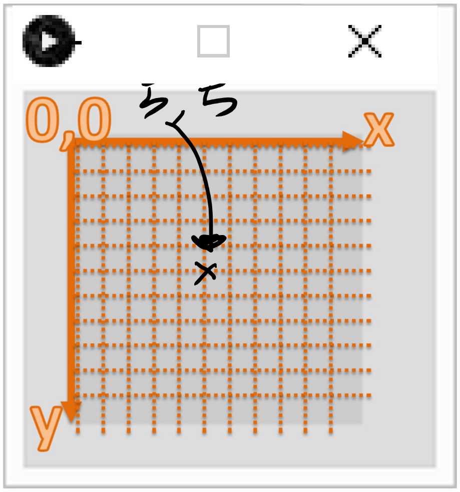
Why can't I just point where I want to draw things???

2.2 The Processing Canvas

There is one more element to the Processing environment to introduce: the *Canvas* (officially called the *Display Window*, but I think Canvas is a better name). If you press *play* in Processing (even without any program typed up), an additional window pops up: the Canvas! You can press *stop* to get rid of the canvas and stop it from running.

Right now, the canvas looks very boring, but – just like a painting canvas – this is where you can be creative and start drawing whatever you want. Unfortunately, you cannot just paint by dragging the Mouse, but you can do it with programming instead!

Notice that the canvas has two sections. The center square, and the outside grey area. The center square is actually the canvas, where drawing happens. The outside is just "padding" to make it fit in the window.



Now, if you want to draw something on the canvas, you need a way to specify *where* to draw things. I suppose you could simply point with your finger or mouse. However, this limits you to placing things once. What if you want to animate something, like, to make a ball move across the screen? And what if this is hard to predict, like in a video game? The position keeps changing, and we need to be able to set it in our program. Because of this, the canvas uses a simple 2D coordinate system (Cartesian coordinates in Euclidean space), like you used in high school for graphing functions. There is an `x` axis, which goes from left to right, and a `Y` axis, which goes from top to bottom. The top-left corner of the screen is `(0,0)`. There are no negative coordinates. Notice the important difference when compared to high-school math: the `Y` coordinate increases as you go down. In regular math, it increases as you go up. Unfortunately, that is due to an old computer standard and is something you will have to remember.

The unit of measure is the pixel, which is the smallest unit of display on your screen. In processing, the canvas default size is 100 by 100 pixels. Don't worry – we will learn how to make a bigger canvas soon. Also, if you draw off the canvas it's no problem, processing just stops at the edge and continues your program.

2.3 Basic Processing Syntax: a command

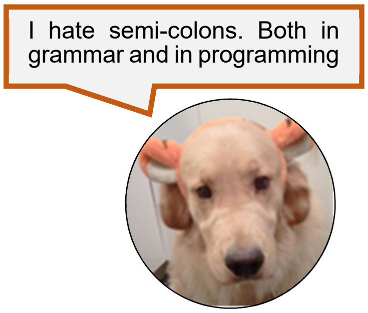
Computers are stupid. To give commands to a computer, we have to follow very strict rules of how to form the command. In programming, these rules are called the ***syntax*** of a language. Even if your command looks reasonable to you, if you do not follow the syntax rules, your program will not work. Think of syntax like grammar, except that if you make even the simplest grammar mistake, the computer has no clue what you tried to say. Throughout this course you will learn a lot of different syntax rules. Let's start with the simple command. Let's learn how to tell processing to draw a line on the canvas.

In order to give a command to a computer, you need to specify what command (e.g., draw a line on the canvas), and you need to give specifics to the command (e.g., where to draw the line on the canvas) – these are called parameters to the command.

Here is the basic syntax for a processing command:

```
command(parameter1, parameter2,...);
```

You give the name of the command, followed by an opening parenthesis (the regular round brackets). Then you add some parameters, with commas between them. You don't put a comma after the last parameter. Then you add a closing parenthesis, and you finish with a semi-colon. The semi-colon means "end of command". You can also add spaces here around the parenthesis and commas, but the formatting is generally like in the example above.



2.4 Your First Processing Command – Draw A Line

Type the following single line of code into the Processing development environment:

```
line(0,0,100,100);
```

and press run. You should see a line being drawn from the top-left corner of the canvas `(0,0)` to the bottom right corner `(100,100)`. Didn't work? Try the following...

- Did you remember the semicolon at the end?
- Did you use the correct brackets? The ( ) and not [ ] or { } or 「」 ?
- Did you type line and not LINE or Line? It is case sensitive (see below)…

Remember that the canvas defaults to being 100 by 100 pixels, and the top left corner is coordinate `(0,0)`. So, the `line` command takes four parameters: the first two are the `x` and `y` coordinate where the line starts, and the second two are the `x` and `y` coordinate of where the line ends. You can imagine the command is like this:

```
line(x1,y1,x2,y2);
```

and you can plug in whatever values you want. Try it, try drawing the line from and to different spots on the canvas.

So, how do you know which parameters (the stuff in brackets after a command) to use and what they mean for a command? Basically, it's confusing, and every command is different.

Having a perfect memory helps intensely!

When is the last time you used the "`Help`" menu in a program? Never? Did you notice that many programs got rid of the help menu (like MS Office)? Well, programmers use on-line help all the time. Reference manuals are very useful for programming because it is not reasonable to expect people to remember all these details.

Open Processing, and click `Help→Reference`. It should open a reference manual in your primary browser. Look for the `line()` command (hint: it's under 2D Primitives) and click on it. At first, this page may look overwhelming, but you will learn to love this reference manual. Notice that it has examples of how to use the command. It has a detailed description, much of which you will not yet understand. That is okay, you should be comfortable with peeking at it and grabbing what you understand. And it even has a Syntax section that explains how to use the command, with explanations of what the parameters are. Take a minute to read the page.

Syntax Errors are *compile-time* errors. This means that Processing refuses to even convert your program into computer language (binary). The error happens before it has a chance to run!

2.5 **Your very first Syntax Errors**

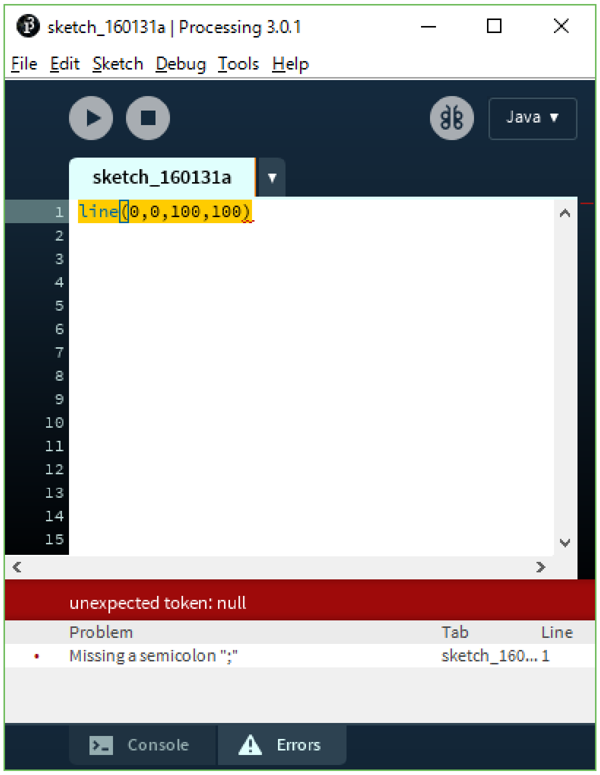
No one avoids mistakes in their syntax (syntax errors). The world's best and most experienced programmers (yes, even the dog!) have regular syntax errors. Syntax errors are not only annoying because you need to know the rules, but our stupid computers are actually really bad at explaining what is wrong with your syntax. To make things worse, for all its awesomeness, Processing is particularly bad at explaining the syntax errors. Try this, type the following command in Processing – this is the command from the previous section, but we removed the semi-colon

```
line(0,0,100,100)
```

When you click *run*, nothing happens. If you look down on the Processing window near the bottom, you will see in the message bar that it says `unexpected token: null`. Now, I'm not sure what engineer thought that this is a nice, descriptive error message, but it's what we're stuck with. In this case, what it's really trying to say is: I expected something here (ahem, a semi colon), but I found nothing!! (`null`). One nice feature is that processing moves the cursor to the spot where it thinks the error is (the cursor is that vertical blinking bar that tells you where you are typing).

Tokens? Are we at an arcade? Is null a new video game?

Processing 3 has a great new feature that tries to help you out with common errors. ***If you click on the "Errors" tab at the bottom of the processing window, you should see a list of errors.*** Here, it suggests that your problem is that you are missing a semi colon, and it even tells you which line (line 1 in this case). If you click on the error, it will even take you to that spot in your program, which is useful as your programs get large.


sketch_160131a | Processing 3.0.1
File Edit Sketch Debug Tools Help
Java ▾
sketch_160131a ▾
Console
Errors

```
line(0,0,100,100)
```

unexpected token: null

Problem	Tab	Line
Missing a semicolon ";"	sketch_160...	1

Fix the syntax error by adding a semi-colon. Your program should work now.

Let's try another one:

```
LiNe(0,0,100,100);
```

Now, Processing gives the syntax error: `The function LiNe(int, int, int, int) does not exist`. We'll learn about those `int`s later, but basically Processing doesn't understand the command `LiNe`. ***Processing (like many languages), is what we call case-sensitive***. This means that it sees upper case and lower case letters as completely different letters. The command `line` will *only* work if all letters are lower case. You'll catch yourself making this error throughout the course. So! If Processing says something doesn't exist, make sure you spelt it correctly AND are using the correct upper and lower case letters.

2.6 More Processing Commands

We need to improve our measly tiny canvas! Biggest canvas is best canvas, right? Whip out our Processing reference (remember, `Help`→`Reference`) and look up the `size()` command. (It's under Environment). Take a quick look through here. Notice how much of it will be over your head (whoa!! Rotate! 3D! what is a renderer?). That's OK – it's a reference, after all, and will include advanced stuff. What you need to home in on is the simple examples and the syntax. The syntax section says:

```
size(w,h);
```

where w is the width of the display in pixels, and h is the height. So, try placing the following command in a clean processing window (if you have stuff in there already, erase it first)

```
size(500,500);
```

and press *play*. You'll now have a nice big canvas! You can even make it bigger! Leave this **at the top of your programs** and you will always have a bigger canvas.

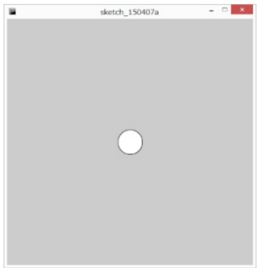
Another great command is `ellipse` – which lets you draw ellipses and circles. Remember, a circle is just an ellipse with the same width and height. Take a peek at the reference page again for `ellipse`, under the 2D Primitives section, and you'll find the following syntax entry (no seriously, go look. This is good practice).

```
ellipse(a, b, c, d);
```

where a and b are the x and y of the ellipse center, and c is the width, and d is the height (`a`,`b`,`c`, and `d` are kind of stupid names for those parameters!!)

So let's draw a circle of diameter 50 at the center of the screen. Remember that our size is now 500 by 500, so the center is `(250,250)`. Erase your processing program and copy the following into processing:

```
size(500,500);
ellipse(250,250,50,50);
```



If all goes well, you should see the canvas to the right with a single circle in the center. Try playing with the parameters of the commands to see what kinds of results you can come up with!

Check your Understanding

2.7 Check your Understanding Part 1 – a quick break!

Usually exercises come at the end of a unit. However, at this stage you need to do some work on your own.

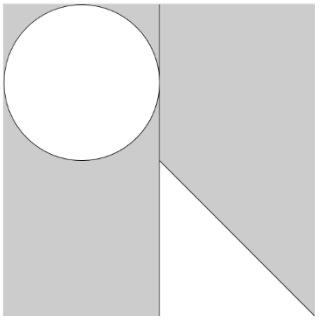
Exercise 1. Investigate the following commands on your own:

a. `point` . This draws a single point. Make a program that uses this command at least three times. This is not a trick question – they will be VERY small and hard to see, so look closely.

b. `rect` . This draws a rectangle. Make a program that draws at least three rectangles.

c. `triangle` . As you may have guessed, this draws a triangle given three points. Make a program that draws at least three triangles.

Exercise 2. Using the commands you learned so far, create the sketch shown to the right. It has 1 line, 1 ellipse, and 1 triangle.



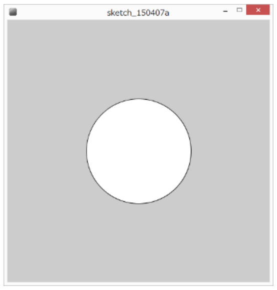
2.8 Paint on top of Paint (order of commands)

Did you know that the Mona Lisa was painted on top of a different portrait?

When you draw on a canvas, you can paint a nice picture and then, paint on top of it to hide whatever was underneath. Because of this painters need to plan ahead – paint the nice broad blue sky first maybe, then add some trees on top. Doing the trees first and then painting in the blue sky is much harder.

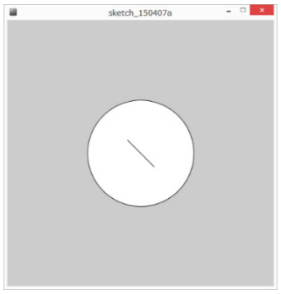
Processing has the same principle. In computer programming, operations are generally run from top to bottom. Later you will learn techniques to make it jump around like crazy, but for now, we start at the top and go down. Try typing and running the following code

```
size(500,500);
line(225,225,275,275);
ellipse(250,250,200,200);
```



You will see the image on the right. Where is the line? Processing didn't forget to draw it. It drew it first, since – top to bottom, the line command comes first – and then drew the circle on top of the line. Try the following variant changing the order of commands:

```
size(500,500);
ellipse(250,250,200,200);
line(225,225,275,275);
```



Now, you see the right image instead.

The order of the commands issued to the program will impact the result. This is not only the case when drawing images, but later, you will see also that this is the case when doing other operations like mathematics. Just remember – top to bottom, one command at a time.

2.9 **Organizing Bigger Scenes: Adding Comments**

Quick! What does the following program do?

```
size(500,500);
ellipse(250,250,300,300);
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);
ellipse(175,225,60,40);
ellipse(325,225,60,40);
ellipse(175,225,15,30);
ellipse(325,225,15,30);
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);
ellipse(250,300,30,30);
```

What do you mean you have no idea? Isn’t it obvious? No? Why not? This is hard to understand because series of commands are hard to understand unless you are the one who wrote them. And even then, give it a few weeks, and you’ll forget too. There are a lot of ways that programmers use to make their code more *readable* and understandable by people. One very important way is the addition of comments.

Comments are English language additions to programs that only serve the purpose of helping a person read the program. The computer completely ignores them.

One type of comment is the **block comment**. This lets you tell Processing that a whole region is text-for-humans and no computers are allowed inside! You can make a block comment by starting it with the `/*` symbols (forward slash then star) and ending with `*/` symbols (star then forward slash). For example

```
/* this is a comment */
```

These are called block comments because they can span many lines and make up a whole block:

```
/* this is a comment, too
   But it keeps on going and going.
   Eric the fish.
   Eric the fruit bat
   Eric the cat
   And don’t forget Eric the kangaroo
*/
```

What if I want to put the */ symbols inside a comment? Will the computer know it’s part of my comment or get confused, thinking I ended my comment?

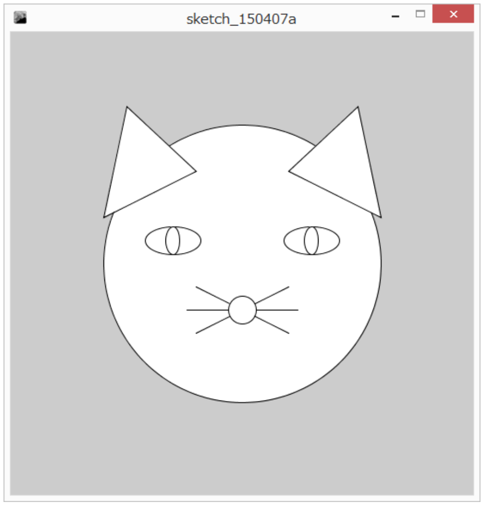
Everything between that start and end character is ignored by the computer.

A very common block comment is the header of your program. At the beginning of a program it is common to give the key information about the program, such as who made it, what the purpose of the program is, etc. ***Header comments are required for all your assignments.***

Let’s add a header comment to our previous program from the beginning of this section.

```
/*******************
* Cat Face! Draw a cat face on the screen
* author: Teo the dog
* version: try #awesome
* purpose: to show how a cat can be drawn
********************/
size(500,500);
ellipse(250,250,300,300);
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);
ellipse(175,225,60,30);
ellipse(325,225,60,30);
ellipse(175,225,15,30);
ellipse(325,225,15,30);
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);
ellipse(250,300,30,30);
```

Now we understand what this program does. The graphical output is shown on the right:



Now, what if I asked you to change the size of the nose, or move the eyes. Which commands would you change? Which `ellipse` command draws each part? You may be good enough to eyeball it (hah!) and figure out which numbers correspond to which piece. But, there's an easier way. Inline comments.

Inline comments are comments that only go from where it starts until the end of that line. It's a simpler comment. You start them with `//` and everything after it on the line is ignored.

```
line(10,10,200,200); // this draws a diagonal line
```

We can add comments when our program is not clear, to help the reader (and ourselves!!) to understand what is going on.

Also notice how I use whitespace (extra lines) to group. Now check out our fully commented version of the program.

```
/********************
* Cat Face! Draw a cat face on the screen
* author: Teo the dog
* version: try #awesome
* purpose: to show how a cat can be drawn
********************/

size(500,500); // make a 500x500 canvas

//draw the head
ellipse(250,250,300,300);

//draw the ears
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);

//draw the eyes
ellipse(175,225,60,30); // left eye
ellipse(175,225,15,30);
ellipse(325,225,60,30); // right eye
ellipse(325,225,15,30);

//whiskers!
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);

// draw the nose. draw after whiskers for nice overlap effect
ellipse(250,300,30,30);
```

Here is a comparison to the original version of the program:

```
size(500,500);
ellipse(250,250,300,300);
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);
ellipse(175,225,60,40);
ellipse(325,225,60,40);
ellipse(175,225,15,30);
ellipse(325,225,15,30);
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);
ellipse(250,300,30,30);
```

```
/*******************
* Cat Face! Draw a cat face on the screen
* author: Teo the dog
* version: try #awesome
* purpose: to show how a cat can be drawn
********************/

size(500,500); // make a 500x500 canvas

//draw the head
ellipse(250,250,300,300);

//draw the ears
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);

//draw the eyes
ellipse(175,225,60,30); // left eye
ellipse(175,225,15,30);
ellipse(325,225,60,30); // right eye
ellipse(325,225,15,30);

//whiskers!
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);

// draw the nose. draw after whiskers for nice overlap effect
ellipse(250,300,30,30);
```

To the computer, the program is identical. But to a programmer, the commented version is much easier to read and work with. ***You will be required to properly comment your assignments.***

One last thing. When do you use a block comment, and when inline? It's basically up to you, but usually inline is for small comments and block for large.

2.10 **Choosing your Paint**

Processing is capable of full color, but for simplicity sake, in this course we generally stick to black and white – or more accurately, greyscale. Processing is capable of a full range from pitch black (well, as black as your screen) to full white. You set the color by specifying it by number. You can think about this number as how much *brightness*. So, a color of `0` is full black. White, however, is not `100` or some other reasonable number. The brightest color is actually `255`.

For Information Only (not testable): You don't really need to understand why, but for the curious, computers *love* powers of two. That's because when you store everything in binary (on and off switches), you end up hitting powers of two all the time. If you are curious about the math, Processing sets aside 8 on or off switches for the greyscale color – this is called an 8 bit number. If you have *n* switches lined up in a row, imagine light switches, then you end up have 2^n possible combinations of those switches. So with 8 switches, you get $2^8 = 256$ combinations. 0...255 is 256 different numbers (include the 0!!).

You can set the paint brush color in processing with the `stroke` command.

```
stroke(gray);
```

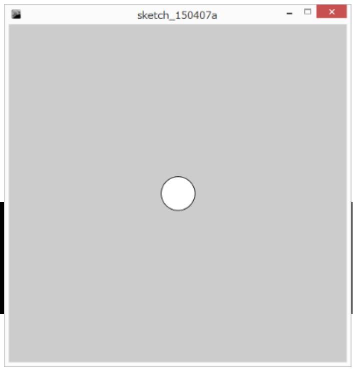
where gray equals the shade from 0...255.

Try the following program

```
stroke(0);
line(0,0,500,500);
```

and this slight variation

```
stroke(255);
line(0,0,500,500);
```



what is the difference? Try it out. You can see that the paint color changed.

Now, try the same comparison, but instead of drawing a line, draw an ellipse:

```
stroke(255); // or stroke(0)
ellipse(250,250,50,50);
```

Notice the difference? In the case with the black paint (color 0), only the outline of the circle changed color and not the center of the circle! So how would we get a black circle?

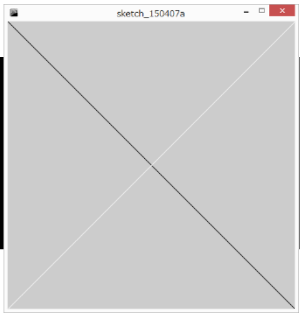
It turns out that processing has two different paints in use at any time. One is the stroke paint (the main brush for outlines). The other is the fill paint (what goes inside a shape).

```
fill(gray) // sets the fill color from 0..255
```

So, we can do a circle with a white outline and a black fill with the following commands.

```
stroke(255);
fill(0);
ellipse(250,250,50,50);
```

You can also switch out your paints at any time. Remember that Processing runs top to bottom. If you use `stroke` and `fill`, they keep fixed for all your drawing commands *until you change them*. For example, you can draw a black line followed by a white line as follows:

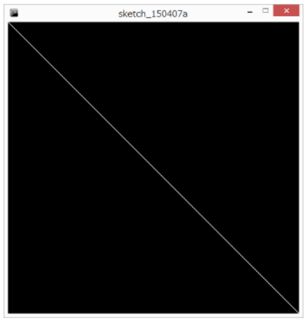


```
stroke(0); // set paint to black
line(0,0,500,500); // first diagonal
stroke(255); // set paint to white
line(500,0,0,500); // second diagonal.
```

There is one thing remaining. How do you reset that annoying ugly grey background? One way would be to draw a rectangle the size of the screen with the stroke and fill color of what you want. Luckily, Processing makes it easier: you can use the `background` command

```
background(gray) // erase the canvas and set to 0..255
```

For example:



```
background(0); // clear to black
stroke(255); // draw with white
line(0,0,500,500);
```

Now, you have all the tools you need to do the full range of colors!!

Check your Understanding

2.11 Check your Understanding: Exercises

Exercise 1. What is syntax? In your own words, try explaining what syntax is to a person with no programming or computer background.

a. How much leeway do you have in your syntax? What happens if you have even a small deviation from the rules?

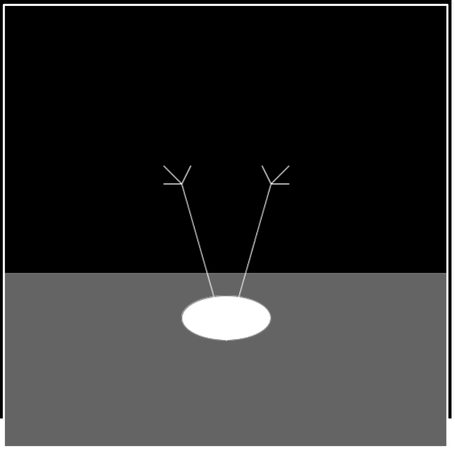
Exercise 2. Make a new Processing program with a 500x500 canvas. Draw a line from the top left corner to the bottom right corner.

a. Did you use 500 or 499 for your end coordinate? Why? Which is correct? (answer: 499).

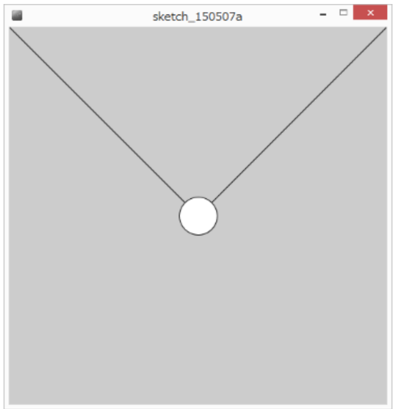
b. If you have a canvas of size `n` by `n` pixels, what is the coordinate of the last pixel in terms of n?

Exercise 3. Here is a program with a series of errors. One way to find the errors is to type the program into Processing and get its help to find them. However, in this case, look at the program yourself and see if you can spot all the errors first. Then, try typing it all up and see if you got them all. There are a lot, 23 in total! The final result should look like the inset at the end of the code.

```
/* broken program
   A bird who caught a very strong worm

canvas(500,500);
Background(0);
Strike(125);
fillColor(100);
rectangle(0,300,499);
fillColor(0);
Circle(250,350,100,50);
strike(256)
Line(250,375,200,200);
Line(200,200,180,180)
Line(200,200;180,200);
Line(200,200,210,180);
line(250,375,300,200);
line(300,200,320,180)
line(300,200,320,200);
line(300,200,290,180);
```



Exercise 4. Write a Processing program as specified below. If you forget how the commands are used, check your class notes for examples or use the reference in the `Help->Reference` menu



a. Make the canvas be 500 by 500 pixels large. (use the `size` command)

b. Draw a perfect circle at the center of the screen, with a diameter of 50 (use the `ellipse` command)

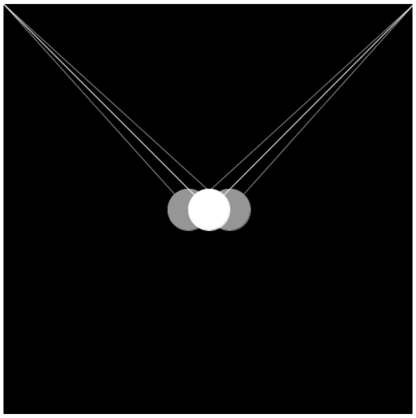
c. Draw a line from the center of the screen to the top left corner (use the `line` command)

d. Draw a line from the center of the screen to the top right corner

e. Make sure the circle is on top of the lines. You should get an image as shown.

f. Make sure you program has a block comment at the top describing your name, the course, and the purpose of the assignment. Also put at least one in-line comment.

Exercise 5. Write a Processing program as specified below. Make sure to have a block comment at the top and proper in-line comments. You will make the diagram as shown in the inset.



a. Set the canvas to be 500 square and clear the background to black

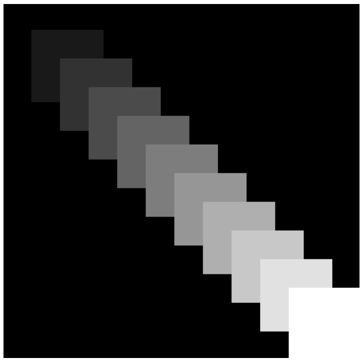
b. Set the stroke and fill colors to solid white, and draw a circle with radius 50 in the center of the screen.

c. Draw white lines from the top corners to the ellipse

d. Set the stroke and fill colors to 150 gray. Draw circles 25 pixels to the right and left of the center circle. Then, draw lines to those

e. You can try adding more circles and increasingly dark colors to create a bigger motion effect.

Exercise 6. Create a processing program to generate the image on the right. There are 11 squares (one is completely black!) so you should be able to calculate the square positions and colors using that knowledge. The square sizes can be whatever looks reasonable.



Exercise 7. Do a web search for “Droodle” – these are popular puzzles that use line drawings to depict a scene. At first, it looks abstract, but once you understand what it is showing, you can see the image.

Pick a few Droodles and try drawing them with processing.

Exercise 8. Read the Processing tutorial on color, which can be found at https://processing.org/tutorials/color/. While we will not use color in this course, color is not hard, and you can quickly learn how to use it. Try modifying some of the above examples to use color.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 3. VARIABLES AND INTEGERS

Summary

In this section, you will…

- Learn how computers can store small bits of information as variables.
- Make your own variables, store information in them, and get information out.
- Get and install the Processing program which enables you to write Processing computer programs.
- Learn the details of syntax surrounding variables.
- Learn about integers.
- See how computers do the basic integer operations of addition, subtraction, multiplication, and division – and see the issues with integer division.
- See how computers use the remainder operation (modulo) with integer division.
- Learn about basic debugging by printing out the contents of variables

Learning Objectives

After finishing this unit, you will be able to write basic Processing programs that use simple integer variables. You will be able to:

- Create a new integer variable and give it a unique name.
- Store information into a variable (e.g., the number 10).
- Get information out of a variable.
- Create a variable and give it an initial value in one line of code.
- Create multiple variables in one line of code.
- Add, subtract, multiply, and divide integers.
- Get the remainder when dividing two integers.
- Apply order of operations to figure out the result of a calculation.
- Use the built-in `println` command to look inside variables for investigating your errors.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

3.1 **Introduction**

What is a variable? We use them all the time in algebra – there, we use a letter or a Greek symbol to represent some number. Some common variables in math are `x`, `y`, `k`, and `l`. Variables are similar in programming, although we use them less for algebra and proofs (yay!). Basically, it is a way to store information. Computers use this all the time. For example, when you log into a website, the website stores your name and personal information in variables, and then uses that information in its processing.

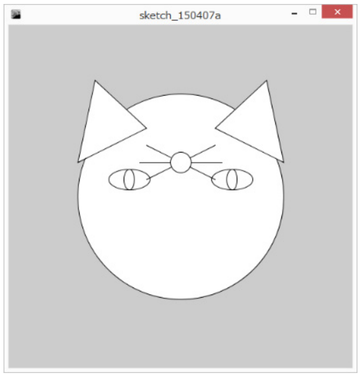
Hi [name], welcome to our site! We see that you come from [city].
People in [city] previously bought the following items from us…

Variables help the programmer write the template and the general case, and fill in the blanks later.

Look at the following program:

```
/********************
* Cat Face! Draw a cat face on the screen
* author: Teo the dog
* version: try #awesome
* purpose: to show how a cat can be drawn
********************/

size(500,500); // make a 500x500 canvas

//draw the head
ellipse(250,250,300,300);

//draw the ears
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);

//draw the eyes
ellipse(175,225,60,30); // left eye
ellipse(175,225,15,30);
ellipse(325,225,60,30); // right eye
ellipse(325,225,15,30);

//whiskers!
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);

// draw the nose. draw after whiskers for nice overlap effect
ellipse(250,300,30,30);
```

Here is our cat program again. Now, what if I wanted to move the whiskers and nose to be a little higher? Let's try 100 pixels higher. To do this we need to change all the `y` coordinates of the whiskers and nose – a real pain in the donkey. Go and do it and press play. You will find that my approximation was off – 100 pixels is way too high and now the cat has a nose on the forehead. So we need to go back and manually do it again, trying a new `y` offset and changing all those values before seeing the result. Wouldn't it be nice if there was a better way?? Well, variables help solve this problem. Remember in algebra we can do things like this:



```
x = 25  // store the # 25 in x
y = 2x   // grab value from x (25), multiply it by 2 (=50),
          store in y
z = 4x + 7 // grab value from x (25), multiply by 4 (=100),
              add 7 (=107), store result in z
```

`y` and `z` are defined relative to `x`. If instead `x` was 30, then y and z are calculated differently, too. It would be nice if we could rewrite our program in a similar way:

`noseCenterY` = 200

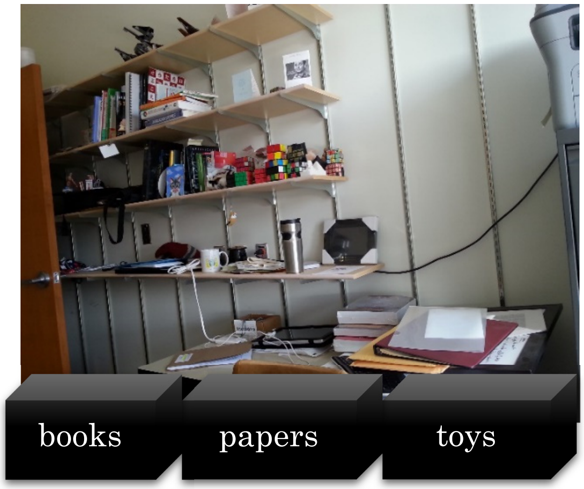
draw straight-out whisker at `noseCenterY`

… (do this for all the drawing commands)

Then, if we want to move the nose, we just change the value stored in our variable, and everything is fixed! All of these are called variables because what is stored in them can change all the time.

3.2 **Variables as Boxes**

You can think of variables as boxes that store information. Imagine you are organizing your office or basement (see the picture). Now, in this imaginary scenario, you are very strict on how you must organize things, so you collect a bunch of boxes



and label them with what goes in them. These labels may be things like books, research papers, and toys. AND, you will be VERY upset if someone mixes things around. No toys in the book box. If this was processing, those boxes are variables, and just like in our scenario, processing is very strict about what goes in each box. When you create a variable (create a box), you need to tell processing about what kind of data you are going to put into it, for example, a number, or some text, or some music! These are called *data types*.

And no exceptions! NEVER put a research paper in a toy box

3.3 **Your First Data Type: the Integer**

What is an integer? (high school math!!). If you remember, it is simply a whole number with no fractional part. The following are integers: 0, 42, -99, and 200. The following are not integers: 3.14, 99.1.

What is the biggest integer? Theoretically, there is none. However, the larger your integer is, the more memory a computer needs to store it, so unfortunately, we cannot have infinitely large integers in Processing since computers do not have infinite memory. In fact, in Processing integers have a pre-defined limit on the range of numbers that can be stored in it

the range -2,147,483,648…2,147,483,647

That is over a 4 billion number range. Clearly, these exact numbers were chosen for obvious reasons. That was sarcasm. Don't bother memorizing these numbers, just have a general sense – about minus 2 billion to about 2 billion.

It' so obvious! Processing uses 32 bits, which has 2^{32} possibilities = 4,294,967,296. If you divide that into positive and negative, divide by two, you get 2,147,483,648. The positive range looks to be one less since 0 is included as a non-negative number. Clearly.

If you need larger or smaller numbers, there are options, but we'll talk about them later.

WHAT?? NO FRACTIONS OR DECIMALS??

That's right – computers are way faster with whole numbers, so we generally resort to those. There are also problems when fractions and whole numbers mix which comes up in a later unit. We'll learn about what to do when we need decimals later.

3.4 Variables in Processing: Syntax

To work with variables in processing, we need to solve three problems:

- how to create a variable
- how to put data into a variable
- how to get data back out of a variable

here is the syntax to create a variable in Processing:

```
variableType variableName;
```

So far, we only know one type, the integer. In Processing, this is converted to the shorthand `int`. So, to create an integer variable called `noseCenterX` to store the `x` coordinate of the cat's nose:

```
int noseCenterX;
```

don't forget the semi colon!

NOTE: Variables must be created (declared) before you use them. (remember that the program runs top to bottom). To avoid this mistake altogether, generally, we put these at the top of our program.

The next problem is how to put data into the variable. The syntax for this is:

```
variableName = data;
```

So in our example, since the cat's nose is centered at the `x` coordinate of 250:

```
noseCenterX = 250;
```

Note: the hand-written data like an actual number, or some text, is called "literals" in programming speak.

So now we can create variables, and we can store data into them, how do we look at them? This is very simple. You just use the variable name anywhere you can use raw data. For example, we can now rewrite the following command:

```
// draw the nose. draw after whiskers for nice overlap effect
ellipse(250,200,30,30);
```

as

```
// draw the nose. draw after whiskers for nice overlap effect
ellipse(noseCenterX,200,30,30);
```

Let’s also add a variable for `noseCenterY`, and update all of the drawing to use our new variables. You end up with the following code:

```
/*******************
* Cat Face! Draw a cat face on the screen
* author: Teo the dog
* version: try #awesome
* purpose: to show how a cat can be drawn
********************/
int noseCenterX;
noseCenterX = 250;
int noseCenterY;
noseCenterY = 300;

size(500,500); // make a 500x500 canvas

//draw the head
ellipse(250,250,300,300);

//draw the ears
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);

//draw the eyes
ellipse(175,225,60,30); // left eye
ellipse(175,225,15,30);
ellipse(325,225,60,30); // right eye
ellipse(325,225,15,30);

//whiskers!
line(noseCenterX,noseCenterY,200,275);
line(noseCenterX,noseCenterY,300,275);
line(noseCenterX,noseCenterY,190,300);
line(noseCenterX,noseCenterY,310,300);
line(noseCenterX,noseCenterY,200,325);
line(noseCenterX,noseCenterY,300,325);

// draw the nose. draw after whiskers for nice overlap effect
ellipse(noseCenterX,noseCenterY,30,30);
```

Now, the nose center drawing uses the information in our variable instead of a hard-coded *literal* number. If you change the `noseCenter` variables around, all the nose commands move nicely. However, there is a problem here!! The whisker ends do not follow along, since we only have the line *starting points* rely on the variable, and not the end points. We'll have to fix this. However, let's learn a few more things about variables first.

3.5 Additional Variable Details

Here is a final list of important points regarding the use of variables. First, when you create a variable, you often want to assign it a value right away. Because of this, you'll often see code like the following:

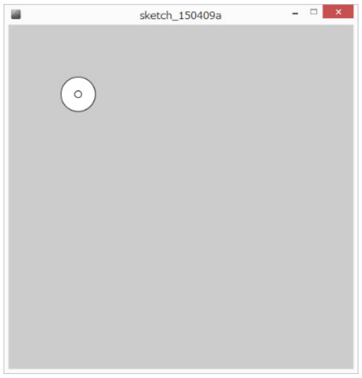
```
int noseCenterX;
noseCenterX = 250;
```

Programmers like shortcuts, so ***there is a clever shortcut for this special case, it lets you create a variable (declare it) and give it a value (assign to it) in one go:***

```
int noseCenterX = 250;
```

With this trick, you can turn two lines of code into one!

Variables are variable: ***you can change variables at any time!!*** And remember that the program runs top to bottom. Check out the following code



```
int circleSize = 50;
ellipse(100,100,circleSize,circleSize);
circleSize = 10;
ellipse(100,100,circleSize,circleSize);
// same as above
```

Even though we type the exact same ellipse command twice, the second one has a different result because the value of `circleSize` changed. Let's quickly step through what happens, going from top to bottom.

1. A new variable is created called `circleSize`, and the number 50 is stored in there.
2. An ellipse is drawn at `(100,100)` of size `(circleSize, circleSize)`. Since `circleSize` currently has the value `50`, it is drawn at size `(50,50)`.
3. The number `10` is stored in `circleSize`. The old value, `50`, is thrown away.
4. An ellipse is drawn at `(100,100)` of size `(circleSize, circleSize)`. Even though this is the same command as in line 2, the result is different because the variable has new data. Now, it is drawn size `(10,10)`.

Further, once a variable has been declared, then you cannot declare it again. Processing gets very confused if you do this! Check out the following code:

```
int circleSize = 50;
ellipse(100,100,circleSize,circleSize);
int circleSize = 10;
ellipse(100,100,circleSize,circleSize); // same as above
```

See the difference with the prior example? Toss it into Processing and see what happens. When you click run, you unfortunately get an error: `Duplicate local variable circleSize`. ***You can’t have two variables with the same name.*** To fix this, you need to remove the `int` in the second `circleSize`, so that you are not trying to create a new one, but instead, just copy a new number to the existing variable. This is a very important distinction – in the first (not-working case) we are asking processing to make a new variable with the same name. In the second case, we are re-using an existing variable and giving it new data (throwing the old data away).

What happens if you create a variable, never put anything into it, but try to use it? For example

```
size(500,500);
int circleSize;
ellipse(100,100,circleSize,circleSize);
```

If you try to run this, Processing will say: `The Local variable circleSize may not have been initialized`. and it won’t run. This makes sense, since what would you expect the circle size to be if you never set a value? ***Variables must be initialized – given a value – before they are used.***

You cannot just name a variable anything, there are rules. However, you have a lot of flexibility for variable names. Generally you should try to pick something nice and descriptive to help you read your code more clearly. Here are the restrictions:

- No spaces! You cannot do: `int circle size;` as it confuses Processing.
- No special characters `!”#%&’()-=^[]{}` – the exception is that underscore is allowed: _ (some people use it as space) and $ is allowed, although no one uses it since it looks funny (e.g., `int a$$e$$e$`);
- Cannot start with a number, but can contain one
  - `int 4peace; // cannot do this!`
  - `int piece4; // this is okay`

Another problem is that some words in Processing already have meaning. These are called `reserved words`. You cannot use these as variable names because they are already in use. For example, you cannot make the following variable:

```
int int;
```

Processing will say: unexpected token: int. This is because it’s trying to add meaning to your variable name and not see it just as a name. Processing also has some of its own variables kicking around, and you cannot use those names either.

You can find whole lists of reserved words and existing variables, but it’s probably not worth your bother to memorize them. Instead, just be aware of the problem in case you stumble across a strange error when making a new variable. Try a similar name and see if the problem goes away, as you may have a conflict.

If you have a number of variables to declare of the same type, ***you can actually declare multiple variables at once.*** Here is the syntax:

```
type variable1, variable2, ….;
```

for example

```
int age1, age2, age3;
```

This is equivalent to

```
int age1;
int age2;
int age3;
```

except that it takes less space.

In addition, you can add in assignments here:

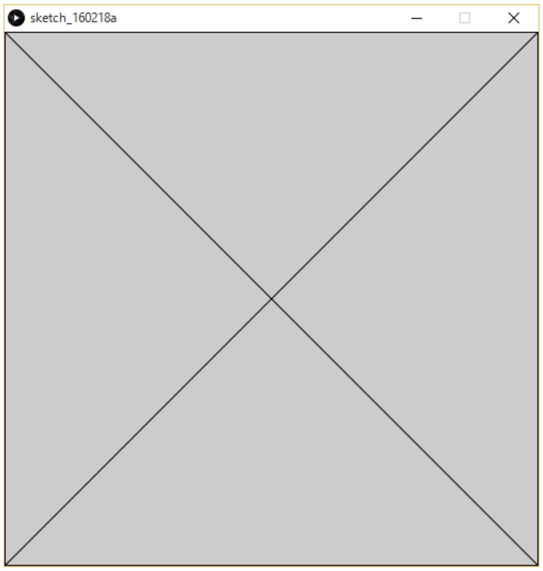
```
int age1 = 17, age2 = 32, age3 = 23;
```

Check your Understanding

3.6 Check your Understanding: Variable Exercises

Exercise 1. Create four integer variables that represent the edges of the screen: `left` and `right` give the x coordinate of the left and right edge, and `top` and `bottom` give the y coordinates of the top and bottom edges.

a. Create the 4 variables, one per line.
b. Set your canvas to size 500,500 and give your variables their initial values. Hint: the right and bottom edges are *not* at 500.
c. Use your variables and the line command to draw two lines to form an X: from the top left to the bottom right, and from the top right to the bottom left.
d. Use your variables and the line command to draw a box around the edge of the screen.
e. Re-write the above program to have all your variables declared and instantiated in a single command.



Exercise 2. The rules say that two variables cannot have the same name, but why does the following code actually work, with multiple variables given the same name?

```
int center = 10;
int radius = 10;
ellipse(center,center,radius,radius);
int Center = 20;
int Radius = 20;
ellipse(Center,Center,Radius,Radius);
```

Exercise 3. Update the cat face example from Section 3.4. Make sure to update your drawing commands to use your new variables as you make them.

a. Add variables for the left eye and right eye (each eye needs two variables) to represent their center locations.
b. Add variables to represent the pupil and eye width and height.
c. Try moving the eyes around and changing their sizes using these variables to make sure you did it correctly.

Exercise 4. Try to do the following without typing them into Processing, as it will immediately tell you the answer. Which of the following variable declaration statements are invalid?

```
a. int cat;
b. Int dog;
c. int mi$$i$$ippi;
d. int dog1, dog2, dog3;
e. int 1mi$$i$$ippi, 2mi$$i$$ippi;
f. int dog, dog;
g. int MEANING_OF_LIFE = 42;
h. int WHY?? = 15;
i. int __SYSTEM_ERR;
j. int &data_;
k. int __data__ = -200;
```

3.7 Integer Operations: addition and subtraction

Luckily, integer operations is a pretty simple topic. This will help us solve our cat whisker problem (above in section 3.4).

Let's start by looking at one of the previous examples, the cat picture. We had the following code for drawing a cat whisker

```
line(250,300,300,325);
```

which we improved by adding named variables to it, as follows:

```
line(noseCenterX,noseCenterY,300,325);
```

Although we can nicely define the center of the nose (the starting point of the whiskers), the ends of the whiskers are still fixed numbers. If you remember, this made the end points of the whiskers stick on the face if we moved the nose.

If we look at the first case above (without the variable name), we can see that the end point of that particular whisker was 50 pixels to the right (`250x` start, `300x` end), and 25 pixels below (`300y` start, `325y` end) the starting point. Once we set the starting point as a variable, we can use integer operations to calculate the other ones.

You can add two numbers together just by putting a plus sign between them. For example, you can do

```
int whiskerEndX;
whiskerEndX = noseCenterX + 50;
```

now `whiskerEndX` will equal to 50 more than `noseCenterX`. In this case, 300.

You can use integer operations anywhere. Here, you don't need a new variable for the whiskers, we can just do the addition right inside the command:

```
line(noseCenterX, noseCenterY, noseCenterX+50,
     noseCenterY+25);
```

NOTE: see how the command here broke across lines because it didn't fit on one? That is okay in processing.

To re-cap, we are drawing a line from `noseCenterX` and `Y`, to `50` pixels more than `X` and `25` pixels more than `Y`. In this case, processing looks inside the variables, gets the values, adds the literal numbers to that value, and uses the result in our line command. There are a lot of steps happening here, but you'll get used to the idea very quickly.

Now, if you change the `noseCenter` variables, that whisker will move along with the rest of the face, since both end points are calculated based on the whisker values.

Let's look at another whisker

```
line(250,300,200,275);
```

In this case, the whisker ends 50 pixels to the left (`250x` start, `200x` end), and 25 pixels above (`300y` start, `275y` end) the start point. We cannot do this with addition, so we need subtraction.

Subtraction in Processing is just as simple as addition. Just use the minus sign: `-`. So, we can rewrite this operation as

```
line(noseCenterX,        noseCenterY,        noseCenterX-50,
noseCenterY-25);
```

If we now update all the whiskers this way, we can move the whole nose around nicely simply by changing the values of the noseCenterX and noseCenterY variables.

If we look at our cat code, we can see another opportunity to use variables. This was included as one of the examples above. As a reminder, here are the eyes:

```
//draw the eyes
ellipse(175,225,60,30); // left eye
ellipse(175,225,15,30);
ellipse(325,225,60,30); // right eye
ellipse(325,225,15,30);
```

Notice how the narrow width of the pupil is 15, the height of the pupil is double that, which is the same as the height of the eye? Also, the width of the eye is 4 times the pupil width. These ratios make the eye look cool, and the pieces just touch nicely. If we were to set the pupil width as a variable, we would need multiplication to calculate the other sizes. For example, if the pupil width was 15, then the height is double the width, and the eye width is four times the pupil width. In our example, we can create a new variable and set it to our pupil width.

```
int pupilWidth = 15;
```

and then plop it into our code.

```
//draw the eyes
ellipse(175,225,60,30); // left eye
ellipse(175,225,pupilWidth,30);
ellipse(325,225,60,30); // right eye
ellipse(325,225,pupilWidth,30);
```

But now to go further we need multiplication.

3.8 Integer operations: Multiplication and Division

Multiplication in processing uses the asterisk – the * symbol (shift-8 on north American keyboards). We can use the multiplication operator on our pupilWidth variable to calculate the remaining widths and heights:

```
//draw the eyes
ellipse(175,225,pupilWidth*4,pupilWidth*2); // left eye
ellipse(175,225,pupilWidth,pupilWidth*2);
ellipse(325,225,pupilWidth*4,pupilWidth*2); // right eye
ellipse(325,225,pupilWidth,pupilWidth*2);
```

As you can see, we just multiply the width by 2 and 4 to get our desired width and heights.

You could imagine that this whole operation could be reversed, and calculated with respect to the eye width (the biggest number) instead of the pupil width (the smallest number). The eye height is half the eye width. The pupil height is also half the eye width. The pupil width is a quarter of the eye width. Do to this, we need division.

Division in Processing is accomplished with the `/` operator. You can envision that this is the symbol used in fractions, like ½. I won't belabor the point, but the above example can be rewritten using division as follows.

```
int eyeWidth = 60;
ellipse(175,225,eyeWidth,eyeWidth/2); // left eye
ellipse(175,225,eyeWidth/4,eyeWidth/2);
ellipse(325,225,eyeWidth,eyeWidth/2); // right eye
ellipse(325,225,eyeWidth/4,eyeWidth/2);
```

Pretty straightforward. Unfortunately, division with integers in Java and Processing is not so simple, and is actually a problem that adds a whole bunch of confusion. Luckily, I'll get into that in a bit and for now, you can use it in this example. But be warned, there are huge caveats.

Go back to the cat example, and try to add more variables.

- Add variables for the center of the cats head. `headCenterX` and `headCenterY`
- Update the head ellipse to use it.
- Update the eye locations to use it.
- Update the ear locations to use it.
- Update the nose center to use it.

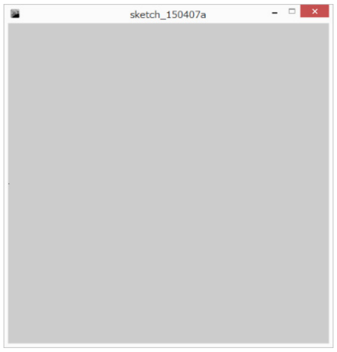
Now, if you do all this work, you have a cat head that you can move around the canvas JUST by changing the center variables. I strongly recommend you try it yourself. Here is my complete solution:

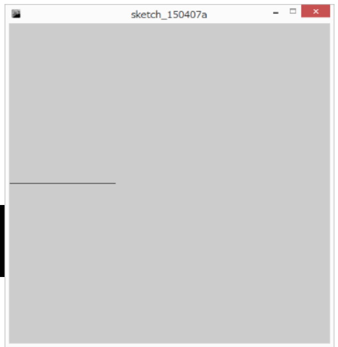
```
/*******************
* Cat Face! Draw a cat face on the screen
* author: Teo the dog
* version: try #awesome
* purpose: to show how a cat can be drawn
********************/

// variables
int headCenterX = 250;
int headCenterY = 250;
int noseSize = 30;
int pupilWidth = 15;
int noseCenterX = headCenterX;
int noseCenterY = headCenterY+50;

size(500,500); // make a 500x500 canvas

//draw the head
ellipse(headCenterX,headCenterY,300,300);

//draw the ears
triangle(headCenterX+125,headCenterY-170,
         headCenterX+50,headCenterY-100,
         headCenterX+150,headCenterY-50);
triangle(headCenterX-125,headCenterY-170,
        headCenterX-50,headCenterY-100,
        headCenterX-150,headCenterY-50);

//draw the eyes
ellipse(headCenterX-75,headCenterY-25,
        pupilWidth*4,pupilWidth*2); // left eye
ellipse(headCenterX-75,headCenterY-25,
        pupilWidth,pupilWidth*2);
ellipse(headCenterX+75,headCenterY-25,
        pupilWidth*4,pupilWidth*2); // right eye
ellipse(headCenterX+75,headCenterY-25,
        pupilWidth,pupilWidth*2);

//whiskers!
line(noseCenterX,noseCenterY,noseCenterX-50,noseCenterY-25);
line(noseCenterX,noseCenterY,noseCenterX+50,noseCenterY-25);
line(noseCenterX,noseCenterY,noseCenterX-60,noseCenterY);
line(noseCenterX,noseCenterY,noseCenterX+60,noseCenterY);
line(noseCenterX,noseCenterY,noseCenterX-50,noseCenterY+25);
line(noseCenterX,noseCenterY,noseCenterX+50,noseCenterY+25);

// draw the nose. draw after whiskers for nice overlap effect
ellipse(noseCenterX,noseCenterY,noseSize,noseSize);
```

3.9 **Advanced Integer Division and Modulo**

Try the following example. Make a program that draws a line across the screen, half way down. Make the line go some percentage across the screen. Set the percentage in a variable, and then calculate how far across to go. The code will look something like this

```
int percent = 33;
int targetX = percent/100*500; // /100 to make percent
line(0,250,targetX,250);
```





type this into Processing, with a 500x500 canvas, and you get the following output: huh? Where's the line? It seems to have not drawn. Let's try doing the calculating by hand: 33/100*500 = 165. If you type 165 into the line code (replace the `targetX` variable with 165), you get the second diagram, with a line one third ways across the screen. Why did the line show up? Why does it work if we calculate it by hand, but not if we ask the computer to do it? They should be the same!

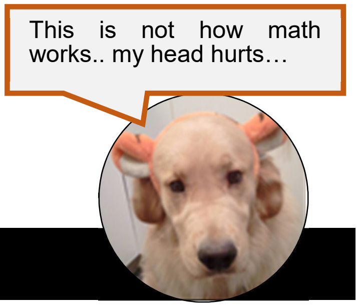
Finally! It's time for an amazing helper tool! Tada! ***Processing has a way for you to peek at data!*** Do you remember the console at the bottom of the Processing window? There is a processing command called print that lets you put data to that screen:

```
println(data); // prints out to the console
```

Let's try it. Modify your above program as follows:

```
int percent = 33;
int targetX = percent/100*500; // /100 to make percent
line(0,250,targetX,250);
println(targetX);
```

Now, in addition to drawing a line, this program will print the value of the `targetX` variable to the console, enabling you to take a peek at what is going on.

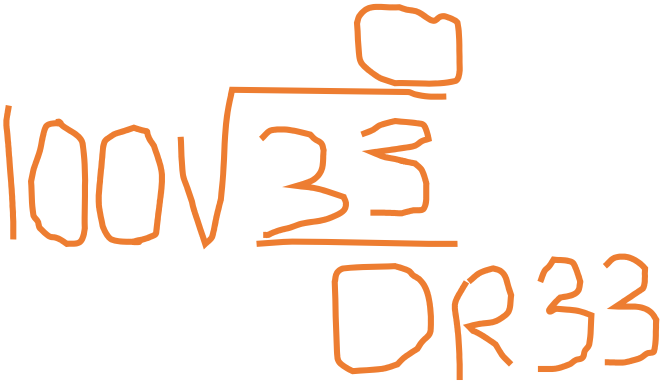


It shows the source of the problem – the `println` tells us that `targetX` is actually set to 0, not to our expected 165. Let's dig a little deeper, try the following command:

```
print(percent/100);
```

Aha! You get 0 again! What did you expect? 0.33?

This is because we are working with integers, and not real numbers. Integer division does not work like you may think! In fact, it works how division worked way back in elementary school. Remember long division? If we did 33/100 in long division, what do we get?



0 remainder 33! Before we learned how to do decimals in school, we had remainders.

In computers, ***Integer division always gives you the answer assuming you want the remainder style, not a fraction result***. What is 1/2? 0 remainder 1. What is 11/3? 3 remainder 2. What is 100/26? 3 remainder 22.

This may seem very strange to you, but as you will see through this course, integer division has a lot of very useful applications. We will learn later how to do the decimal style with real numbers, as you may naturally expect. In all these cases, the remainder is thrown away, and the basic result (0, 3, 3, etc.) are kept. So how do we get the remainder? We use the modulo operator.

Modulo is a very unusual operation that you will learn a lot about in mathematics classes. Basically, it gives you the remainder when two numbers are divided. Let's look at one of the above examples, 11/3. Remember, 11/3 is 3 remainder 2.

```
println(11/3); // 3 is output
```

To get the remainder, we replace the / symbol with `%`, the percent sign.

```
println(11%3) // the number 2 is output, the remainder
```

So if 11/3 is 3 remainder 2, we get the 3 with division, and we get the 2 (remainder) with modulo. You should practice this. What is the remainder when you do 10/2? 5/2? 11/3? We will end up using remainder a lot later. For now, there is very little we can do with it that is fun and exciting, so we'll come back to it.

3.10 Order of Operations

Processing math operations follow standard order of operations from mathematics. What is the result of the following command?

```
println(3+2*6/3%4);
```

The answer depends on the order that you do the math. In high school, you learned a particular order that math happens – luckily, this follows here. Multiplication (and division and remainder) happens first, and then addition and subtraction:

- 3+2*6/3%4
- → 3+12/3%4
- → 3+4%4
- → 3+0
- → 3

If you do the addition first, you get the wrong answer.

Sometimes, the order of operations can be quite unclear. In this case, you can always use brackets to be sure and to enforce what you mean – just like in high school math:

```
println( 3+ (2*6/3)%4 );
```

The code inside the brackets happen first.

Check your Understanding

3.11 Check Your Understanding: Exercises

Exercise 1. In the following code, what does b equal at the end? 20 or 10? Hint – read top to bottom, and data is always copied. There are no links or such happening

```
int a;
int b;
a = 20;
b = a;
a = 10;
```

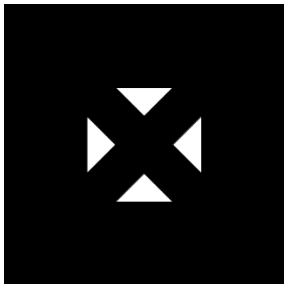
Exercise 2. Create a simple program that calculates the volume of shipping containers and determines how many boxes of sand can fit into the container if they are emptied in. A common standard size is 12m by 2.6m by 2.9m.

a. Create variables for the length, width, and height of the shipping container. You should assume cm for the unit.
b. Calculate the volume by multiplying these together and store it in a variable.
c. Create variables for the length, width, and height of the shipping box. E.g., a U-Haul moving box is 45cm by 45cm by 41cm.
d. Calculate the volume of sand in the box and store it in a variable
e. Calculate how many whole boxes of sand can fit in the container and store the result in the container. Use the integer division.
f. Since we are using integer division there is no fractional component. Calculate the remainder from the above division. What does this number tell you?
g. Use the `println` command to output your results in any fashion that you like.

Exercise 3. Using your knowledge of order of operations, by hand calculate the result from the following calculations. Do you get the same result as posted?

a. 6+5*2+10 = 26
b. 10*10+10*10 = 200
c. (1+1)*(2+2) = 8
d. 6%2%2%2 = 0
e. 1%2 = 1
f. 1+2-3*4/5%6 = 1

Exercise 4. This is a tedious exercise, but will really give you a lot of practice with the coordinate system and using variables. You will make a program to generate the crosshairs on the right, based on three initial variables: `spacing`, `crossX`, and `crossY`. The x and y is the center of the cross hairs, and the spacing defines how large it is.



a. Create the top center triangle first. Create 6 variables, two per point. For example, `t1LeftX` and `t1LeftY` to represent triangle 1's left point X and Y position.
   i. For the x coordinates, the center triangle point is at the `crossX`. The left and right are offset by `crossX` plus or minus the `spacing`.
   ii. For the y coordinates, the center triangle point is offset from `crossY` by the spacing. The top two points are offset by `2*spacing`. Hint: the y coordinates of the top two points are

the same, so once one is calculated, copy that result into the other variable.

b. Then, create the bottom triangle in a similar fashion. Notice that the x coordinates of the bottom triangle are the same as the top. Do not re-calculate them, but instead copy them from the first triangle.
c. Do similar operations for the left and right triangles.
d. Try changing the size and position of your crosshair to make sure your variables and calculations are working properly!

Exercise 5. Copy-paste the cat example from the unit, where you can change variables to move it around. You will add one more functionality: scale. By changing a variable, try to make it so you can shrink or grow the head. There are a lot of ways to do this, my steps is just one way.

a. Create a variable called `scale`, and set it to the size of the head (300 in this case). Calculate the other sizes (nose, pupil, etc.) based on this. E. g., the pupil is currently one twentieth of the head size and the nose is 1 tenth
b. Here is the tricky part: all of your offsets, for the whiskers, eye locations, etc., will need to change. You already have the numbers on the 300 scale for all components, but will need to think about how to have them scale automatically.
    i. Hint: your numbers (such as the whiskers being 25 or 50 offset) are on the 300 scale. How can you change scales? Careful, with integer division you cannot do straight percentages.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

UNIT 4. CODING STYLE AND STANDARDS

Summary

In this section, you will…

- Learn about the importance of coding style and standards
- See how Processing provides some mechanisms to help you
- Get exposed to one reasonable style

Learning Objectives

After finishing this unit, you will be able to …

- Choose meaningful variable names that make your code more readable.
- Create *named constants* that cannot change once set, as a safety feature.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

4.1 Introduction

Much of computer programming involves writing computer programs to get work done – to make the computer do what you want it to do. However, as you will soon learn as your assignments get harder, most of your time is actually spent re-reading, editing, and ***debugging your code: removing errors and problems that stop your program from working as you expect it should.*** In fact, in professional programming much more time is spent modifying and debugging code than is spent writing it. By the end of this course, you will already be at that level. In addition, a great deal of the energy of professional programmers is spent on reading each other's code. You will find yourself increasingly reading your own code from some time ago.

All of these things point to the importance of having clear computer code that is easy to understand and read. Coding style and standards are developed precisely for this reason. While it may seem that this is about making code readable by others – and therefore does not pertain to this class – at your current level, coding style and standards are very important to help *you* be more efficient. If your code is easy to read and understand then you are more likely to see problems, and, you have more brain power left for thinking of other potential problems.

By far, the most common problem I see from students in my office hours in this course, is that there is a difference between what the student *thinks* their code says, and what it *actually* says. My usual strategy is to simply logically go through the code with the student, at which point the student notices the problem and can fix it: I do nothing but to read out what is on the screen. The code style and standards introduced in this course are carefully chosen to help avoid these problems, and help you, the students, be more effective and efficient.

This is a very short unit with only a little bit of information given. However, as new techniques arise coding style and standards will continue to pop up.

We have already learned some ways to make our code more readable. We use comments when possible to add additional explanations. We use indentation properly to help your eyes line up the code blocks and to easily know where they start and end. We have also been trying to use good variable names, but I will talk a little more about that.

4.2 Meaningful Variable Names

One of the most common issue for new programmers is the use of terrible variable names. For example,

```
int a; // bad, no meaning
int a2; // even worse!
```

We do use some variable names that are single letter, but this is only acceptable because it is standardized and actually comes from math. For example, `i`, `j`, and `k` are common iterators for indices, and `x,y,z` are common for coordinates. Only use single-letter variable names when it is very clear from the context what it means, e.g., when implementing a math formula.

Good variable names describe your code and what is happening, and results in you requiring less commenting. For example, I often see cases like:

```
int p1S; // player 1 score
int p2S; // player 2 score
```

This forces the programmer to memorize what `p1S` and `p2S` mean, and they may have to look back at the comment several times as they work through the code. Instead, the comments and the issue can be easily avoided completely by just choosing good variable names.

```
int player1Score;
int player2Score;
```

Sometimes, variable names have to follow standards. Many companies or software projects provide you with their variable naming scheme and you need to follow it so that everyone is using the same system. For example, some places start short-term variables with an underscore (e.g., `int _data`). In this class we do not enforce any particular variable naming scheme, but you must use descriptive variable names – use your common sense here, and err on the side of being too descriptive instead of not descriptive enough. The examples used in class are about where you should be aiming.

Also, you'll notice that for my variable names ***I use a technique called camel case***. This is where the first word starts with a lower case, but the following words start with upper case to make it easier to read, even though there are no spaces. `thisIsCamelCase`. This is called camel case because you can imagine a camel from the side view with its head down (the first letter), and the bumps on its back being the capitals.

Let's do another example. Look at the following example code. What does the calculation mean? Clearly we are multiplying numbers, but what numbers, and what does the result tell us?

```
int resultA = 100*5*26
int resultB = 52*5*26;
int resultC = 88*5*26;
```

What does this mean? What purpose does this serve? Well, in this case it is a summer cottage industry calculating seasonal costs. The first result is hydro costs, the second is maintenance costs, and he third is firewood costs. If we choose better variable names it is a little easier to parse:

```
int hydroCost = 100*5*26
int maintCost = 52*5*26;
int woodCost = 88*5*26;
```

Now the variables are *self-commenting* as in our example above. However, those other numbers are still confusing. What do they mean?

Some programmers call these confusing literals, these seemingly-random numbers hanging in calculations, *magic numbers.* They are called magic numbers because it looks like magic; we have no understanding of how it works or why it is there. I once had a professor who would deduct 10% from my assignment for every magic number found!

To improve this, and to avoid magic numbers, we should start using named constants.

4.3 Named Constants

A ***named constant is a value or piece of information which we guarantee will not change while the program is running***, e.g. the length of a business season, amount of sales tax, etc. For example, imagine that we re-write the above example as follows:

```
int hydroCost = 100*DAYS_PER_WEEK*WEEKS_PER_YEAR;
int maintCost = 52*DAYS_PER_WEEK*WEEKS_PER_YEAR;
int woodCost = 88*DAYS_PER_WEEK*WEEKS_PER_YEAR;
```

Although this is more verbose, and requires a lot more typing, this is much easier to read and understand. This is self-commenting code. Except for the remaining magic numbers (100, 52, and 88), I do not need text or comments to explain the calculation.

Notice that the new variables are in ALL CAPS. ***We often use ALL_CAPS_WITH_UNDERSCORES_FOR_SPACES when doing constant values***, this is a naming convention that is quite universal. That is, we only make the variables ALL_CAPS to signal to the programmer that the variable is a constant one; it has no impact on how the computer treats the code. Naming for constants is in contrast to regular changing variables which we use camel case for as previously described.

In case you are slightly confused here, all of these variables would need to be declared and set prior to using them, earlier in the program.

Just to re-cap: some variables here are ALL_CAPS to signify that it is a constant, it will not change. This includes how many days are in a week, and how many days are in a year. Other variables are using camelCase which signifies that they are regular variables that can change, such as the amount of wood cost daily to provide heat, which will change with the seasons.

Let's compare this code to the original:

```
// original code
int resultA = 100*5*26
int resultB = 52*5*26;
int resultC = 88*5*26;

// improved code
int hydroCost = 100*DAYS_PER_WEEK*WEEKS_PER_YEAR;
int maintCost = 52*DAYS_PER_WEEK*WEEKS_PER_YEAR;
int woodCost = 88*DAYS_PER_WEEK*WEEKS_PER_YEAR;
```

Although the new improved code is much longer, it is self-documenting – the calculation is obvious.

Another important benefit of using named constants is that important numbers are only maintained in one location. Let's say the industry changed how many weeks per year they were open. In this case, instead of hunting through the code to find where that number is used (the number 26), and potentially changing other 26s as well, we simply update the variable value. As long as that named constant is changed, the calculation will be fixed wherever the variable is used. This avoids a common bug where a programmer forgets to fix a few obscure instances, for example.

Although we can signify named constants with our ALL_CAPS convention, Processing actually has a mechanism for specifying named constants. This is particularly useful as Processing will not let anyone change the constant, not even by accident.

To create a named constant in Processing, you use the following syntax. This is essentially the same as the regular syntax to create a variable, except we add the final keyword.

```
final type VARIABLE_NAME; // a named constant
```

For example:

```
final int WEEKS_PER_YEAR = 26;
```

Final variables can only be set once, and the can never change. This is to avoid mistakes where you accidentally change something that is not meant to change. Examine the following code:

```
final int WEEKS_PER_YEAR;
WEEKS_PER_YEAR = 26;
WEEKS_PER_YEAR = 0; // ← illegal because already set
```

In this case, Processing will not yet you do the second assignment operation. Try it out to see what happens.

You may be thinking that one good place to use named constants is to set your canvas size. Then you can use the constant in the `size` command, and throughout your program. ***Unfortunately, that doesn't work in this case!!*** It is a great idea, and should work this way. However, there is a quirk in how Processing converts your program to data, which makes this not allowable. Sorry. To clarify, you cannot do this:

```
final CANVAS_SIZE = 500;
size(CANVAS_SIZE, CANVAS_SIZE);
```

The parameters to the `size` command unfortunately must be literal.

For Information Only (not testable): There is an additional advantage to using constants. If you use a constant in a calculation, your computer will examine if it can take short cuts. That is, it will see if it can calculate the result once, before the program is even run. For example, in the above program, `DAYS_PER_WEEK * WEEKS_PER_YEAR` is a constant calculation that the computer does not need to re-calculate every time. For very advanced programs that use a lot of constants (like video games), this savings can add up to be a significant optimization. Use named constants whenever you can.

Note: for your assignments, you will be marked on your use of style and standards. You need

- Reasonable variable names
- Consistent and good indentation
- Reasonable comments where needed (err on the side of too many)
-

Check your Understanding

4.4 Check Your Understanding: Exercises

Exercise 1. Choose better names for the following variables so that comments are no longer needed:

```
// calculate the average age of my polo team
int a; // first player age
int b; // second player age
int c; // third player age
int d = (a+b+c)/3;
```

Exercise 2. This small piece of code is used by an air conditioning contractor to calculate the volume of air in a rectangular house, to choose which unit to install. In this case, the house is 10m by 30m, and there are three floors, each is 3m tall. Create variables and named constants, and add comments, to make the code more readable.

```
int volume = 10*30*3*3;
```

Exercise 3. The following code calculates the weight of a pack of pens. The size of a pack of pens never changes – it always contains 10 pens. Also, the weight is fixed. Update the following code to used named constants for both of these. There is a catch, what is it? Type it in processing to see.

```
int pens = 10;
int penWeight_g = 10;
int totalWeight_g = pens * penWeight_g;
pens = pens - 1; // remove one from the box
```

Exercise 4. The following code is a very boring program – it takes some kind of key number (in this case, 90210), and uses it to generate a unique scene. All the quantities, the colors, and the size and location of the ellipse, are generated based on this number. First, type the program into processing and get it working (try your phone number!). Then, fix the program using comments, good variable names, and named constants, to be more readable.

```
size(500,500);
int a = 90210;
int b = a%256;
int c = (a*13)%256;
int d = a%(500/2);
int e = a%500;
```

```
        int f = (a*13)%500;
        d = max(10, d);
        e = max(0, e);
        f = max(0, f);
        background(b);
        stroke(c);
        fill(c);
        ellipse(e, f, d, d);
```

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

UNIT 5. ACTIVE PROCESSING

Summary

This is a difficult unit – we finally move away from boring calculation programs and start to have programs that animate and you can interact with. These are called *active* programs in the Processing lingo. In this section, you will…

- Move away from static programs toward interactive, dynamic programs
- Learn about code blocks, logical units or groups of programming commands
- See how a variable, once created, only works in some parts of the program: it has a scope within which it's valid
- Learn how to get the mouse position
- Learn some new commands for finding out which of two numbers is larger or smaller

Learning Objectives

After finishing this unit, you will be able to …

- Create an active program that can change while it is running
- Create animations where the drawing changes as time passes
- Use the mouse location as input into your program
- Calculate which of two numbers is larger or smaller

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

5.1 Introduction

So far, you have been using what processing calls *static sketches* – your programs run straight through once and you see the final result. This is cool, but it's time to start doing something more exciting: *active sketches*!!!!

Active sketches start up, and then keep on running and running and running, repeating the program and making changes, until you press stop. This is exciting as it opens up a whole new world of programming, such as animations and programs that act based on your mouse and keyboard actions. So far, you use Processing just by typing commands in. To switch over to active mode, there are no settings in Processing – instead, it's how you write your program.

First, here is a magic line of computer code that draws a line from the center of the screen to a random location. We don't yet have all the tools necessary to properly learn the `random` command, so for now, please be patient with just typing this in and not fully understanding it. Assuming that our canvas is 500 by 500, here is a complete program:

```
size(500,500);
line(250,250,random(500),random(500));
```

run this program, and you will see a random line from the center of the screen to some spot. Run it again. And again... and you will get a different line each time. That is each time random is called, it generates a random number for the `line` command (in this case, from 0 to 499, excluding 500, but we'll learn that in more detail later). This illustrates the limitation of the static program – it only runs once. To move forward, let's learn about active sketches.

5.2 Active Sketches

To move from static to active sketches, we just need to type in our program a little differently. In an active sketch, you actually specify two chunks of computer code: code that runs just one time at the start of your program, and code that gets repeated. We call these two fundamental regions "blocks" of code. ***There is a setup region – this is code that runs once at the beginning of your program.*** This is kind of like the static programs you have been creating already. The second region is called ***a draw region. This code gets run over and over and over again.*** In fact, since computers are so fast, Processing tries to run it 60 times every single second. You can change this speed, but that's beyond the scope of this course.

Active sketches require some new syntax. To mark off these sections of your program – the run once setup code, and the repeating draw code – you need to learn about code blocks. Processing marks off a block of code by sandwiching it between two curly brackets: `{` and `}`. For example:

```
{
  // this is a code block
}
```

Be careful not to use the square brackets `[ ]` or the round ones `( )` (and especially not the triangle ones `< >`) as they mean different things in Processing.

Notice how I indented the code inside the block? This is a good practice to get into as it makes your program easier to read. You'll get a lot of practice with this, especially as we start using blocks all over the place later on. Hint: ***The Processing editor has a feature to automatically do your indenting. It is hidden in one of the menus – can you find it?***

You don't usually create a code block just by itself, like in the above example, but it gets attached to something. In this case, we need to give these code blocks names so processing knows which one is which. We simply put the name in front with some special syntax.

```
void setup()
{
  // run once
}

void draw()
{
   // run 60 times every second
}
```

What exactly does this mean? For now, just memorize this syntax. The word void means *empty* or *nothing*, meaning the block doesn't generate any data for others to use. The empty brackets `()` means that the block doesn't require data when it is started to get going. This is confusing, and you will learn more about it later in the course.

Advanced: You have already seen many commands, like `line`, `ellipse`, etc., that require parameters (data) to run, e.g., screen coordinates. When we create blocks, they can likewise be configured to take data (we'll learn much later in the course). We haven't yet seen any commands that give you data (except `random`, in passing), but blocks can likewise return (result in, and pass along) data to be used elsewhere. The syntax for creating a block in processing enables you to specify parameters and return data, even though we won't be using it for a while. Here is the general syntax for setting up a block, which we will learn later:

```
resultType blockName(paramType1 paramName1, type2 name2,…)
{
   // block of code
}
```

Our setup and draw blocks do not provide any results (like `line`!) and do not take any parameters, so we can set them up with the `void` return type and no parameters.

Try typing the setup and draw code from the previous page into Processing and make sure it runs. However, since there is no code inside the blocks, nothing should happen but seeing an empty canvas. Make sure there are no errors.

So what exactly is happening here?

- 1) Processing runs the `setup` block one time. This is exactly like your static Processing programs you have been using throughout the course.
- 2) Processing runs the `draw` block. It runs every command top to bottom.
- 3) Processing waits until it's time to draw again, and then runs the `draw` block again. This repeats at about 60 times a second (60 Hertz) until you close the program or until an error is encountered.

Advanced: Why does processing need to wait before drawing again? 60 frames per second (FPS) may seem very fast, but from the perspective of a computer that measures time in billionths of a second, the time between frames is very long. For 60 FPS, there is a delay of about 0.0166 seconds between drawing each frame. On the billionths of a second time scale, that is equivalent to 16 million clock ticks, in which time the computer can get a whole boatload of work done. So, it needs to wait before drawing again, and will probably check your email, do some networking, etc.

Let's rewrite our earlier random line example, from the beginning of the unit, in active Processing. Let's put everything into the run-once setup block.

```
void setup()
{
   size(500,500);
   line(250,250,random(500),random(500));
}
void draw()
{
}
```

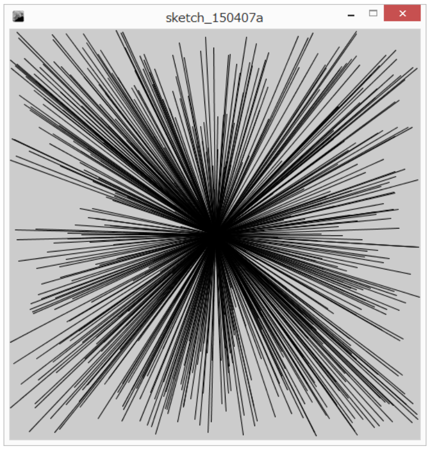
This should run as expected. The setup code runs once only – this acts the same as your static sketch did before. The repeating draw block is still empty.

Now, let's see what happens if we move the line command into the draw block. Keep the size command in the setup block because we only want it to run once, at the start of the program.

```
void setup()
{
   size(500,500);
}
void draw()
{
   line(250,250,random(500),random(500));
}
```

Yay!! We now have an animated program. Let's step through what is happening.

- 1) Processing runs the `setup` block one time. The `size` command gets run and we have a 500x500 canvas.
- 2) Processing runs the `draw` block one time. A line is drawn from 250x250 to some random location.
- 3) Processing waits until it's time to draw again.
- 4) Processing runs the `draw` block again. A line is drawn from 250x250 to a *new* random location. Repeat.



After just a few seconds, we see something like on the right.

We can now put any code we want into the draw block, and it will play over and over and over again.

NOTE: in active Processing, all commands generally must go inside a block. The only exception is creating variables, which we will get to below.

5.3 **Variables and Blocks**

Now that we have blocks, variables get a little more complicated – where do you define them?

Let's try the following example: what do you reasonably expect to happen?

```
void setup()
{
   size(500,500);
}
void draw()
{
   int linePosition = 0;
   line(250,0,linePosition,linePosition);
   linePosition = linePosition + 1;
}
```

Let's step through this. The `setup` block is run one time, and the canvas size is set. The `draw` block is run. The variable `linePosition` is created and set to 0. The line is drawn from `250,0` to `linePosition`, `linePosition` which is 0x0. `linePosition` is increased by 1. So far so good. Processing waits until it is time to draw again, and then runs the `draw` block again. Next time, since we changed `linePosition`, will it draw to `1,1`? And then `2,2`? Run it to see.

If you try this out, it doesn't work – we only get a line from `250,0` → `0,0`. Apparently, the value in `linePosition` doesn't increase as we expect. What the heck is going on?

Variables and blocks have some very important properties. ***If you define a variable inside a block, it only exists until the block is finished. You can do it, but once the block is over, the variable actually gets destroyed.*** This is particularly annoying for the draw block, since it runs 60 times a second. Any variable you create in there gets destroyed and re-created EACH TIME, losing its data from before. So each time through in our example, `linePosition` gets created, initialized to 0, used, increased by 1 (to a value of 1), and then destroyed, to be re-created on the next draw. It doesn't hold its value.

What happens if we move the variable declaration to the setup block instead? This way, perhaps the variable can be created only once, but used many times?

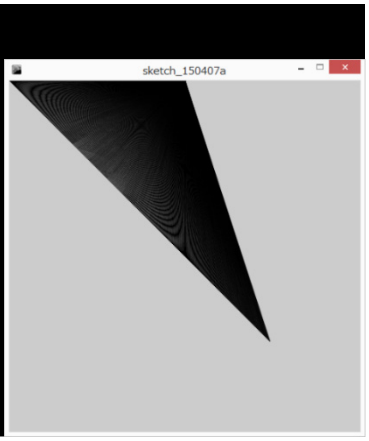
```
void setup()
{
   size(500,500);
   int linePosition = 0;
}
void draw()
{
   line(250,0,linePosition,linePosition);
   linePosition = linePosition + 1;
}
```

Try to run this. Processing complains and raises an error. It says: `Cannot find anything named “linePosition”`. That’s strange, we created it right there! This raises another property of variables and blocks: ***A variable created inside one block is only visible from within that block – other blocks cannot see them!*** This is called a ***scope rule*** – it defines the area where a variable is visible. So, the variable we created inside setup cannot be used or seen inside draw, since its scope is limited to setup.

The combination of the two above rules regarding variables and scopes is a little complex, but they will stick with you throughout the course – pay special attention to them. A) variables only exist within the bloc they were created, and get destroyed at the end. B) variables created inside one block is only visible from within that block.

In this case, the solution is to move our variables outside our blocks to the top of our program. These are called ***global variables***. This way, they are created outside the blocks. They are created when the program starts. They exist until the program ends – they do not get destroyed. They are also visible to all our blocks:

```
int linePosition = 0;
void setup()
{
   size(500,500);
}
void draw()
{
   line(250,0,linePosition,linePosition);
   linePosition = linePosition + 1;
}
```



Cool! It works! We draw a line from that spot across the whole diagonal. You may notice that instead of being solid black, the region has a funny curvy wavy pattern. This is as a result of *anti-aliasing*, an advanced graphics technique that is way beyond this course! Feel free to ask your instructor about it.

One more thing. Why does the line build up a black region across the canvas, instead of just moving along? What do you do if you want an actual line moving across – and not leave that trail behind? Well, just like a real canvas, if you keep drawing, your paint adds up. If you want an animation with something moving, you need to clear the canvas each time. Each time you draw, you need to actually erase the old paint first. We learned this back in chapter 2. The `background` command erases to a color, so you can add this at the beginning of the `draw` block to start over with a fresh clean canvas each time! (try `background(127);`);

Where do you put the background command? If you put background at the end of the drawing, what happens? It erases all the paint *after* you do your drawing but before showing it on screen, so you end up seeing nothing. Put the `background` command at the beginning of the `draw` block to start with a fresh canvas. Here is the updated program

```
int linePosition = 0;
void setup()
{
   size(500,500);
}
void draw()
{
   line(250,0,linePosition,linePosition);
   linePosition = linePosition + 1;
}
```

Now is a good time to re-evaluate the order that things happen in, since we have global variables. ***When you press run, this is what happens, in the order specified:***

- 1) the global variables are created and initialized
- 2) the setup block is run one time
- 3) the draw block is run. Wait and repeat, so that it is run at 60 times per second

5.4 Processing Globals – mouse position

In addition to making your own global variables, Processing provides some global variables for you that you can use in your program. One of the best ones (!!) is mouse position, which you can access through the following global variables:

```
// current position of the mouse pointer, x and y
mouseX
mouseY
// the previous position of the mouse, last time we drew
pmouseX
```

```
pmouseY
```

Note that these are variables that you can use, and are not commands. What is the difference? Commands do work underneath the hood. You also call commands with parenthesis ( ) at the end of them. These are variables, so you can just read their current value like any other variable.

Note: The mouse position variables store the position of the mouse at the beginning of the draw block. It may have already moved by the time you draw something! This usually doesn't matter, but may explain some strange effects you may see.

You can use these variables at any time. Try the following sample:

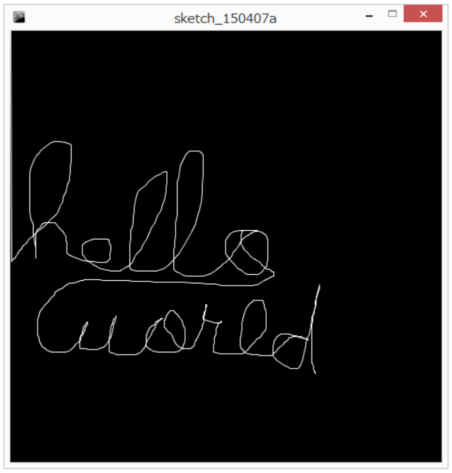
```
void setup()
{
  size(500,500);
}
void draw()
{
  background(0);
  stroke(255);
  line(0,0,mouseX,mouseY);
}
```

Very cool! You can make a line draw to the mouse cursor! Each time it runs, it clears the background to black, sets the color to white, and draws from `0,0` to the mouse location. Try moving the mouse around while the program is running.

Here is another quick example. Here, if we draw a line from the previous position to the current one, what do we get?

```
void setup()
{
  size(500,500);
}
void draw()
{
  background(0);
  stroke(255);
  line(pmouseX,pmouseY,mouseX,mouseY);
}
```

This one is a lot of fun, you have to really try it – it draws a line behind the mouse, like you're being followed by a tiny worm. What happens if you remove the background command – so that we don't erase the paint each time? Move that command to the setup block so that you only clear the background once. You have a painting program!!



5.5 Example – hold the roof up!

Let's draw a line across the screen that slowly falls to the bottom of the screen. However, you can hold it up with your mouse: that is, as the line falls, if it hits the mouse, it will stop. There are a few pieces to this. First, let's make an active sketch where the line falls.

We need a global variable for the current line position. This has to be global because we need to remember the line position between draws, so it can draw at a new position each time as it falls.

```
int linePosition = 0;
```

Then, in the draw block, let's increase its position by one, and draw a line across the screen at that Y coordinate. Don't forget to erase the canvas

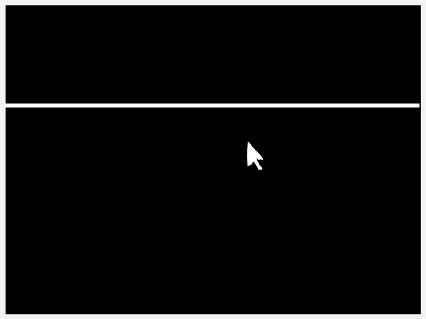
```
void draw()
{
  background(255);
  linePosition = linePosition + 1;
  line(0,linePosition,500,linePosition);
}
```

Now we have a line that starts at a y of 0, and increases by 1 each frame, to fall down the screen. Now for the second part: how can we push it back up with the mouse? Well, what we need to do is to compare the mouse y coordinate with the line. If the mouse is above the line – if the y is smaller – then move the line to that point. Luckily we have some helper functions that we can use here. We have

```
max(a,b); // gives you the biggest of two numbers
min(a,b); // gives you the smallest of two numbers
```

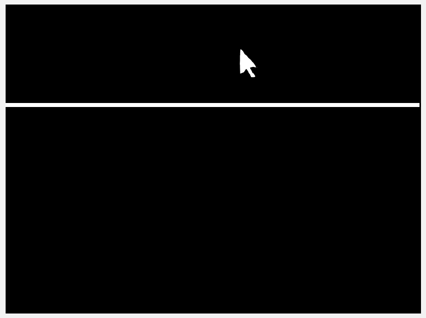
This is a little tricky, but using `min`, we can find out which is smaller – the line position or the mouse – and set the line to that.

```
linePosition = min(linePosition, mouseY);
```



Think through this for the second. The min command takes the linePosition and the mouseY, and returns whichever is smaller. We have two basic cases, shown in diagrams here. If the linePosition is smaller than the mouseY (higher up, as in the first image), then min will give us the line position. So in this case, the line position actually doesn't change from the min command, and the line just keeps falling. As expected.

```
linePosition = min(linePosition, mouseY);
// if line position is smaller than mouseY,
// then this essentially becomes
linePosition = linePosition; // no change!
```



In the other case, if the mouseY is smaller than the linePosition, then the min command returns mouseY. So, the line position jumps to the mouse location. In this case, the line cannot fall anymore, because each time, the line gets brought back to the mouse position.

```
linePosition = min(linePosition, mouseY);
// if mouseY is smaller or equal to line position,
// then this essentially becomes
linePosition = mouseY; // move the line to the mouse
```

This solves the exercise! You can now hold up the line. Here is my final code:

```
int linePosition = 0;
void setup()
{
  size(500,500);
}

void draw()
```

```
{
   background(255);
   linePosition = linePosition + 1;
   linePosition = min(mouseY, linePosition);
   line(0,linePosition,500,linePosition);
}
```

Now is a good time to look closely at this program. Do you know what order things happen in? Do you remember exactly what happens, step by step? This course will only start to get more complicated, so now is the time to ensure that you understand the mechanics here – you need to know what happens, in what order, piece by piece.

5.6 Example – make the cat face active

Let's try to update our earlier cat face to be an active sketch. One great thing we did was to make the cat nose and whiskers all draw relative to some variable values. Don't remember? Re-visit Unit 3 for a quick refresher. What happens if we link these variables to the mouse coordinates?

First, type up the last cat example again from Section 3.8. Run it to make sure it works.

To make the program global, you need to break the program into the `draw` and `setup` blocks. In this case, keep the variables global. Remember that no commands except for the variable declarations can be outside the blocks. How do you decide what goes into the `setup` and `draw`? Remember: do once, goes in setup. Draw each time, goes in draw. In this case, the setup only needs the canvas size setting.

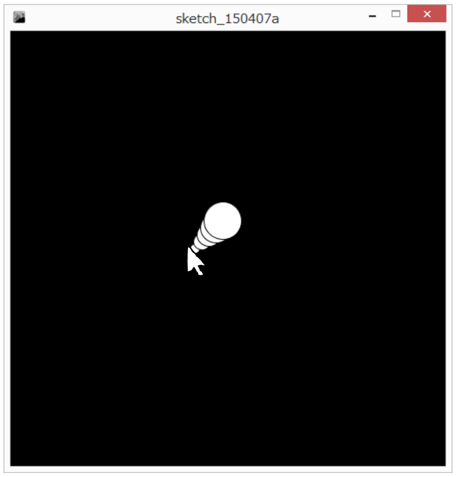
Next, since we are re-drawing the cat rapidly, make sure to clear the canvas (probably to black?) on each draw. This way, if we move the cat, we won't have the old paint kicking around. Now, run your program to ensure it works.

Finally, let's animate the mouse. Currently, the nose center of the cat is linked to the global variables `noseCenterX` and `noseCenterY`, which are set to static values at the start of the program. To animate the cat, instead update the nose center based on the mouse position. At the beginning of the `draw` block, copy the mouse position into the nose center variables. Now you have an animated cat face! Try running it!

Just in case you hit a snag, here is my final cat-face code. You should try linking the mouse to other things, such as eye position, size, etc. Play around with this.

```
/*******************
 * Cat Face! Draw a cat face on the screen
 * author: Teo the dog
 * version: try #awesome
 * purpose: to show how a cat can be drawn
 ********************/

// variables
int headCenterX = 250;
int headCenterY = 250;
int noseSize = 30;
int pupilWidth = 15;
int noseCenterX = headCenterX;
int noseCenterY = headCenterY+50;

void setup()
{
  size(500, 500); // make a 500x500 canvas
}

void draw()
{
  background(0);
  // copy the mouse position into the nose position
  noseCenterX = mouseX;
  noseCenterY = mouseY;

  //draw the head
  ellipse(headCenterX, headCenterY, 300, 300);

  //draw the ears
  triangle(headCenterX+125, headCenterY-170,
    headCenterX+50, headCenterY-100,
    headCenterX+150, headCenterY-50);
  triangle(headCenterX-125, headCenterY-170,
    headCenterX-50, headCenterY-100,
    headCenterX-150, headCenterY-50);

  //draw the eyes
  ellipse(headCenterX-75, headCenterY-25,
    pupilWidth*4, pupilWidth*2); // left eye
  ellipse(headCenterX-75, headCenterY-25,
    pupilWidth, pupilWidth*2);
  ellipse(headCenterX+75, headCenterY-25,
    pupilWidth*4, pupilWidth*2); // right eye
  ellipse(headCenterX+75, headCenterY-25,
    pupilWidth, pupilWidth*2);

  //whiskers!
  line(noseCenterX, noseCenterY,
       noseCenterX-50, noseCenterY-25);
  line(noseCenterX, noseCenterY,
       noseCenterX+50, noseCenterY-25);
  line(noseCenterX, noseCenterY,
       noseCenterX-60, noseCenterY);
  line(noseCenterX, noseCenterY,
       noseCenterX+60, noseCenterY);
  line(noseCenterX, noseCenterY,
       noseCenterX-50, noseCenterY+25);
  line(noseCenterX, noseCenterY,
       noseCenterX+50, noseCenterY+25);

  // draw the nose after whiskers for nice overlap effect
  ellipse(noseCenterX, noseCenterY, noseSize, noseSize);
}
```

5.7 **Example – mouse bubbles**



Let’s make bubbles come out from the mouse cursor. Like in the inset, except animated – they come up out of the mouse and then pop, and another one comes up.

How do you attack this problem? The bubble is size 0 at the mouse pointer and slowly grows to some maximum size (say, 50). As it grows, it moves up (-y) and right (+x).

First, just think about the static cases. This is a common technique for tackling a larger problem. If we can solve the basic problem in a few of the static cases, it makes seeing the animation solution a little easier. Let’s draw a bubble of size 0 at the mouse pointer:

```
ellipse(mouseX,mouseY,0,0);
```

clearly this is useless. But, what bubble is next? It's a size `1` bubble, with `+1` `x` and `-1` `y` in comparison to this one.

```
ellipse(mouseX+1,mouseY-1,1,1);
```

What bubble is next? Size 2

```
ellipse(mouseX+2,mouseY-2,2,2);
```

Do you see the pattern? At each step, we add to x the step amount, subtract it from y, and make the ellipse that size. Look at them lined up

```
ellipse(mouseX,mouseY,0,0);
ellipse(mouseX+1,mouseY-1,1,1);
ellipse(mouseX+2,mouseY-2,2,2);
```

Why don't we replace those numbers with a variable and see how it looks?

```
int step = 0;
ellipse(mouseX+step, mouseY-step, step, step);
```

Now, to draw the next bubble, we increase step by one and use the exact same ellipse code.

```
step = step + 1;
ellipse(mouseX+step,mouseY-step, step, step); // same above
```

If we look at all three, then we can see a pattern

```
int step = 0;
ellipse(mouseX+step, mouseY-step, step, step);
step = step + 1;
ellipse(mouseX+step, mouseY-step, step, step); // same above
step = step + 1;
ellipse(mouseX+step, mouseY-step, step, step); // same above
```

This is nice because the ellipse command is identical, and all that we do is change the `step` variable. In this case, we don't want to draw all 50 bubbles at once, we want them to animate. Also, copy-pasting this 50 times is tedious and not a good use of time. So what can we do? Since we know that `draw` runs repeatedly, if we make `step` a global variable, we can increase the step size each time draw runs.

This way, each draw gives us one bubble. Then it gets bigger for the next draw, and so on. It also gets further away. Here is how it may work.

```
//globals
int step = 0;
…
// in draw
ellipse(mouseX+step, mouseY-step, step, step);
step = step + 1;
```

Try coding this up. yes! We have the animation! So what's the next problem?

Step just keeps getting bigger and bigger… but we want it to reach size 50 and then start over with a new bubble.

There are several ways to solve this. However, one of the simplest ways is to use a technique that we actually already learned, but unfortunately, is not very obvious at all. We can do this using modulo (the remainder function). Let me explain…

Let's say that you have a number, *n*, and you divide it by, say, 4. What possible results can you get? What remainders can you get? You should try calculating out n from 1 to 20 and see what remainder you get. Here are the first 8.

n	0	1	2	3	4	5	6	7	8
n/4	0	0	0	0	1	1	1	1	2
n%4	0	1	2	3	0	1	2	3	0

Do you notice anything special about the remainder? What possible remainders can we get? If we do n/4, can we get remainder 5? Why not?

Remainders of *n*/*m* are always in the range from 0 to m-1. In this case, if you divide a number by 4, it is always in the range 0...3. Once it hits 4, the division result gets bigger, and the remainder re-starts at zero. This is quite confusing, think on it a minute.

So how does this help us? We can use modulo to make our step size loop. If we do the following:

```
step = step + 1;
step = step % 50; // remainder when /50
```

Then step will go from, 0,1,2,…,49. When it hits 50, the remainder is 0 again. You may have noticed that I can write the above two lines in one single line:

```
step = (step+1)%50; // same as above.
```

And it works! Step gets bigger, but once it hits 50, it starts at 0 again. We're done this example! Here is my final code:

```
int step = 0;

void setup()
{
  size(500, 500);
}
void draw()
{
  background(0);
  ellipse(mouseX+step, mouseY-step, step, step);
  step = (step + 1)%50;
}
```

Make sure you take time to really understand what is going on here. There are only a few lines of code, but this is suddenly a lot more complex than what you have been seeing in previous units. Now is the time to get help with this if you're not following, since things are just going to get harder from here.

Check your Understanding

5.8 Check Your Understanding: Exercises

Exercise 1. Create an active Processing program that draws a circle directly under the mouse as it moves around. Make the circle have a radius of 20, and make it white against a black background. There should be no trails left behind the mouse as it moves around.

a. Update this example so that the width of the ellipse is 1 tenth the `mouseX` coordinate, and the height is 1 20th of the `mouseY`.

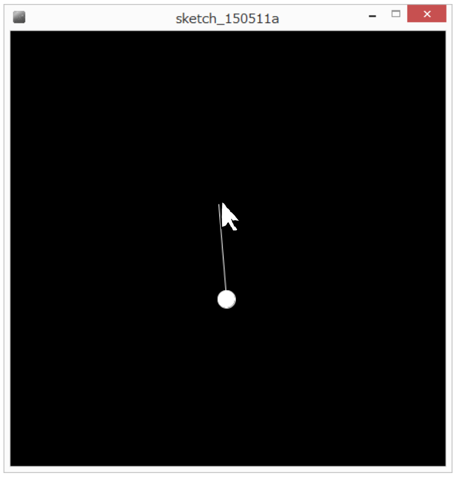
Exercise 2. Create an active Processing program that visualizes the remainder (modulo or mod) operator (%). You will do this by making a block that follows the mouse, but the block's size will be the remainder when the x coordinate is divided by 100. Specifically:

a. Create the `setup` and `draw` blocks, and resize the canvas to 500x500 in the setup block.
b. Create the following global integer constants (use "final" to declare

them) and set them to reasonable greyscale values: `BG_COLOR`, `BOX_COLOR`

c. In the `draw` block, create the following local variables and set them to the current mouse coordinates: `boxX` (set to `mouseX`), `boxY` (set to `mouseY`)
d. Create a local variable `boxSize` and set it temporarily to size 30.
e. Use the `rect` command to draw the box using `boxX`, `boxY`, and `boxSize`. Don't forget to set the fill color.
f. Run your program and make sure that the box follows the mouse around. Notice that the box touches the mouse at the top left corner.
g. Next, make the box centered on the mouse. Offset `boxX` and `boxY` by half the size of the box. That is, subtract `boxSize/2` from `mouseX` and from `mouseY` when setting `boxX` and `boxY`. This will make the box centered on the mouse. Be sure to use the `boxSize` variable and not literal numbers.
h. Run your program – it should now follow the mouse but be centered on the mouse.
i. Now, to visualize mod, set the box size to the remainder when you divide the `mouseX` coordinate by 100. Remember that you can get the remainder by using the `%` operator. The size of the box will now change to values from 0 to 99.
j. Run your program. You should now see what the remainder looks like as you move the mouse from left to right across the x axis – it gets larger until it reaches its maximum size, then resets to size 0, and repeats!
k. Finally, after calculating `boxSize`, but before using it to calculate `boxX` and `boxY`, make sure that the size is never less than 10. Use either the `min` or the `max` function to do this.

Exercise 3. Create a program that simulates a ball hanging from the mouse on an elastic band.



a. Create an empty active Processing program with setup and draw blocks, and a canvas of 500x500.
b. Create global integer variables to keep track of the ball's position. Also create variables to set the ball's color and size. The ball can start initially at the center of the canvas.
c. Make the draw block draw the ball and clear the canvas each frame. Use global variables. Your program should run now although not much

will happen.

d. In each iteration of the draw block, make the ball move toward the mouse. This is tricky, you do it as follows. Do the X and Y separately.

   i. Calculate how far the ball is from the mouse in terms of X: create a new local integer variable called `diffX` and set it to `mouseX - ballX`.

   ii. If we add `diffX` to `ballX` (`ballX = ballX + diffX`) then the ball will move to the mouse. Try the algebra on a piece of paper and you will see it.

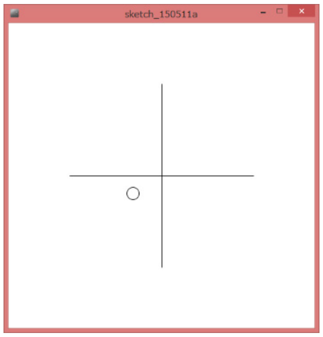
   iii. Instead, add 1/10th of the difference to `ballX`. Do this in one line, by setting `ballX` to `ballX` plus `diffX` divided by 10.

   iv. Do this for the Y coordinate as well in the same way

e. Draw a line from the ball to the mouse to simulate an elastic.

f. Now, add gravity! In each frame, simply add 10 pixels to the Y. The ball will fall, and, will stop falling when it tries to move 10 pixels toward the mouse but is pulled down 10 pixels by gravity.

Exercise 4. Create a program where the mouse acts like a joystick to drive a ball on the screen. In the picture shown you can see a cross hairs, and a ball. This program will work as follows: if the mouse is at the crosshairs, the ball doesn't move. If the mouse is to the right or left, the ball moves right or left. If the mouse is above or below the center, the ball moves up or down. The distance of the mouse from the center determines how fast the ball moves.



a. First, draw the crosshairs. The exact dimensions do not matter as long as they center in the middle of the screen.

b. Create some global variables to remember where the ball is. Start the ball at the center of the screen. Don't forget to draw the ball1

c. Calculate how much the ball should move in the x and y directions separately and store in the variables `moveX` and `moveY`.

   i. For x, you use `mouseX-250`. If the mouse is at 250 (the screen center), then we get 0, no movement! If the mouse is left of 250, you get negative movement! If it is to the right, you get positive movement.

   ii. Do the same for Y.

d. Add the `moveX` and Y to the ball position.

e. Run your program, the ball should move. But – Whoa! It flies off the screen! Use min and max to make sure the ball stays within visible range

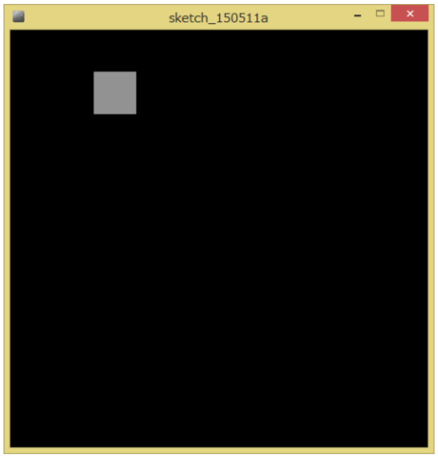
   i. Use one check for the right side of the screen, and one for the left. It is easy to get `min` and `max` backward here, so try an example on paper.

ii. Use another set of checks for the top and bottom of the screen

iii. Can you figure out how to do it so that the whole ball stays on the screen and not just half? The basic solution works off of the ball center, so it goes over the edge…

f. Finally, the ball moves way too fast. How can you make it move slower? Just use division on the move speeds.

Exercise 5. Let's write a processing program that uses some clever math to make visual effects. This program will have two distinct features: a rectangle will snap to a grid under the mouse (move along the grid as you move the mouse instead of being right under the mouse), and, the color will follow a simple formula that makes it look unpredictable.



a. Create a clean active Java program, with a canvas sized 500

b. You will make a rectangle snap to a grid 10 by 10 (100 places). Make a global called `gridSize` and set it to 10. In the draw loop, make a local variable called `cellSize` (how big each cell is) and store the size of each cell. Hint: screen size divided by grid size.

c. Calculate the top left corner for a rectangle to draw. This corner will have to snap to this grid. That is, it can only be `0,0`, or `cellsize*i, cellsize*j` for some integers `i` and `j`. You will find the nearest grid point to the top left of the mouse cursor. There is a trick to it.

d. Integer division always throws away the fraction part. If you divide the mouse coordinate by the cell size, it will tell you how many cells along the mouse is. Do this separately for `x` and `y` and store in `int cellX, cellY`. This is not the coordinates on the screen, but rather, the cell number

e. Convert from the cell numbers to screen coordinates. This is easy. Just multiply the cell number by the cell size. Cell 0 gives you 0, cell 1 gives `cellSize`, and cell 2 gives `2*cellSize`

f. Draw a rectangle at this cell

g. Now you should be done the drawing part. Set some simple color for now and debug up to this point before moving along. Next let's work on the color.

h. Create a global integer called `runningColor` to keep track of the last color we used. Then, each frame calculate the color as follows. This is not something you should develop an intuition for, it's just a silly bit of math to make the color act interestingly as you move the mouse around.

i. Take the product of the `mouseX` and `runningColor` and add `mouseY` to it. Then modulo the result by 256, and multiply by -1. Store this final result as the new `runningColor`.

j. Before setting the current fill and stroke color to running color, make a copy and cap it above zero in case it happened to go negative. Don't store the capped number in `runningColor`, if its negative let it stay that way for next time.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 6. USER-DEFINED FUNCTIONS PART 1

Summary

Here we learn a generalized extension of making your own named code blocks, like the `draw` and `setup` block. This is not actually a hard unit, but is a little lengthy because of the code examples. You will learn to make your own commands. In this section, you will…

- Lean about user-defined functions and how they form part of a program.
- Learn how user-defined functions help you make larger programs that are more manageable.
- See how user-defined functions complicate variable scope rules.
- Learn about how user-defined functions can call each other, to further simplify your programs.
- Learn a technique called top-down programming to help you attack difficult problems

Learning Objectives

After finishing this unit, you will be able to …

- Create your own user-defined functions.
- Use your custom functions within other functions.
- Use your custom functions to avoid repeating code.
- Apply the top-down technique to solve problems.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

6.1 Introduction

Professional projects can be hundreds of thousands or even millions of lines long.

As your programs get larger and larger, and more complex, you (very!) soon reach the limit of the human brain to keep it all organized and effective. While some people believe that they are good enough to keep their programs in one big long block (like ours so far), there is a lot of research that says they're wrong, and highlights the benefits of taking a more structured approach to programming. In addition, you will find yourself repeating the same code over and over again. We need to learn new techniques that help us manage our larger programs, organize them, and re-use code whenever possible to save work. Enter user-defined functions! To be clear, you are the user here, who will be defining your own functions.

User-defined functions are a way to create your own commands. So far, you have used many commands like `rect`, `ellipse`, and so on. Actually, in the last unit you learned how to make functions when you made your draw and setup block. User-defined functions is just a simple but powerful extension of what you already learned.

Advanced: You may hear about functions, procedures, subroutines, or methods, which all refer to a very similar concept. To make things more confusing, there isn't always perfect agreement on what they mean. The most important difference is between methods and functions, procedures, or subroutines. Methods are specifically functions that are linked to an object, which ends up giving it special properties. Other things you may hear include functions *returning* (providing) data and procedures not *returning* data. At this point, don't worry about these subtle differences.

I have opinions about cat faces, but probably shouldn't share them publicly...

One of the largest programs we have actually written so far is our good old cat face. Are you tired of it yet? Don't worry, we're almost done with it. If we look at the cat face example, not only do we notice that the `draw` block gets really long, but also that it breaks down nicely into clear units of work. These units include drawing a head, eyes, ears, etc.

Here is the whole code again.

```
/*******************
 * Cat Face! Draw a cat face on the screen
 * author: Teo the dog
 * version: try #awesome
 * purpose: to show how a cat can be drawn
 *******************/
```

```
// variables
int headCenterX = 250;
int headCenterY = 250;
int noseSize = 30;
int pupilWidth = 15;
int noseCenterX = headCenterX;
int noseCenterY = headCenterY+50;

void setup()
{
  size(500, 500); // make a 500x500 canvas
}

void draw()
{
  background(0);
  // copy the mouse position into the nose position
  noseCenterX = mouseX;
  noseCenterY = mouseY;

  //draw the head
  ellipse(headCenterX, headCenterY, 300, 300);

  //draw the ears
  triangle(headCenterX+125, headCenterY-170,
    headCenterX+50, headCenterY-100,
    headCenterX+150, headCenterY-50);
  triangle(headCenterX-125, headCenterY-170,
    headCenterX-50, headCenterY-100,
    headCenterX-150, headCenterY-50);

  //draw the eyes
  ellipse(headCenterX-75, headCenterY-25,
    pupilWidth*4, pupilWidth*2); // left eye
  ellipse(headCenterX-75, headCenterY-25,
    pupilWidth, pupilWidth*2);
  ellipse(headCenterX+75, headCenterY-25,
    pupilWidth*4, pupilWidth*2); // right eye
  ellipse(headCenterX+75, headCenterY-25,
    pupilWidth, pupilWidth*2);
  //whiskers!
  line(noseCenterX, noseCenterY,
       noseCenterX-50, noseCenterY-25);
  line(noseCenterX, noseCenterY,
       noseCenterX+50, noseCenterY-25);
  line(noseCenterX, noseCenterY,
       noseCenterX-60, noseCenterY);
  line(noseCenterX, noseCenterY,
       noseCenterX+60, noseCenterY);
  line(noseCenterX, noseCenterY,
       noseCenterX-50, noseCenterY+25);
  line(noseCenterX, noseCenterY,
       noseCenterX+50, noseCenterY+25);

  // draw the nose after whiskers for nice overlap effect
  ellipse(noseCenterX, noseCenterY, noseSize, noseSize);
}
```

Now, let's imagine that we had new commands for all the components of the cat face. We could really simplify the `draw` block if we used these commands. To be clear, at this point I am just making up hypothetical command names – this code will not run.

```
void draw()
{
 background(0);
 // copy the mouse position into the nose position
 noseCenterX = mouseX;
 noseCenterY = mouseY;
 drawCatHead();
 drawCatEars();
 drawCatEyes();
 drawCatWhiskers();
 // draw the nose after whiskers for nice overlap effect
 drawCatNose();
}
```

If this worked it would have a great deal of benefit. One, it enables us to see an overview of the draw block without scrolling through pages. Two, we have a lot fewer comments! Since we have the command names, it is obvious what we are doing. The only comment left explains the nose / whiskers draw order. At this point, we assume that the code that actually does the work – the lines, ellipses, etc. is somewhere else.

But how can we create such commands? We need to learn the new syntax!

6.2 Basic Functions: Syntax

Luckily for us, you have already learned the syntax to create a new function – you did this for the `draw` and `setup` blocks.

To create a new function (a new command) we need two basic things:

- function name: they keyword used to invoke (use) the function. E.g., `ellipse` is a function name
- some code: the processing code to run every time the function is invoked (called).

With this in hand, we can create a new function using the following syntax:

```
void functionName()
{
   …//code;
}
```

The `void` keyword means that the function does not create any data result, and the empty brackets () means the function does not require any data to run. We will learn more about that later in the course.

Look familiar? See? We already learned the syntax. The main difference between our existing draw and setup commands, and these new user-defined functions, is that draw and setup already have defined roles, and Processing knows when to use them. Your new commands will not be used at all until you do it yourself!

The first line, which currently starts with `void`, is called the ***function header***. The function block is called the ***function body***.

You must put this code outside of your startup and draw blocks, along with your globals. You can place them before or after those blocks, it's up to you and ends up being a matter of style. If you try to put them inside an existing function, it will not work.

Whatever we put as `functionName` we then can use in our program to run the command. For example, the following is a perfectly valid user-defined function to draw a centered circle that takes up half the screen:

```
void drawCenteredCircle()
{
  int canvasSize = 500; // should be in a global!
  int circleSize = canvasSize/2;
  ellipse(canvasSize/2, canvasSize/2,
          circleSize, circleSize);
}
```

But now that we've created this command, how can we use it? Simple: we just use it like any other command, by typing its name with brackets. In this case:

```
drawCenteredCircle();
```

To be clear, here is how the whole program may look.

```
void drawCircle()
{
  int canvasSize = 500; // should be in a global!
  int circleSize = canvasSize/2;
  ellipse(canvasSize/2, canvasSize/2,
          circleSize, circleSize);
}

void setup()
{
  size(500,500);
}

void draw()
{
  drawCircle();
}
```

You can call your new `drawCircle` command from anywhere, and as often as you like. From the perspective of the draw block, this is just another command just like `line`, `ellipse`, etc.

Now, let's go back and look at the cat example from the beginning of this unit. We already have the `draw` block made with hypothetical commands. To finish the example, we can create the new commands and copy the code over. The only thing to consider is that your new functions can work with the global variables exactly like the draw and setup block. That is, nothing has changed. Try it yourself before looking at the solution posted here.

```
/********************
 * Cat Face! Draw a cat face on the screen
 * author: Teo the dog
 * version: try #awesome
 * purpose: to show how a cat can be drawn
 ********************/

// variables
int headCenterX = 250;
int headCenterY = 250;
int noseSize = 30;
int pupilWidth = 15;
int noseCenterX = headCenterX;
int noseCenterY = headCenterY+50;

void setup()
{
  size(500, 500); // make a 500x500 canvas
}

void drawCatHead()
{
  ellipse(headCenterX, headCenterY, 300, 300);
}

void drawCatEars()
{
  triangle(headCenterX+125, headCenterY-170,
    headCenterX+50, headCenterY-100,
    headCenterX+150, headCenterY-50);
  triangle(headCenterX-125, headCenterY-170,
    headCenterX-50, headCenterY-100,
    headCenterX-150, headCenterY-50);
}

void drawCatEyes()
{
  ellipse(headCenterX-75, headCenterY-25,
    pupilWidth*4, pupilWidth*2); // left eye
  ellipse(headCenterX-75, headCenterY-25,
    pupilWidth, pupilWidth*2);
  ellipse(headCenterX+75, headCenterY-25,
    pupilWidth*4, pupilWidth*2); // right eye
  ellipse(headCenterX+75, headCenterY-25,
    pupilWidth, pupilWidth*2);
}

void drawCatWhiskers()
{
  line(noseCenterX, noseCenterY,
    noseCenterX-50, noseCenterY-25);
  line(noseCenterX, noseCenterY,
    noseCenterX+50, noseCenterY-25);
  line(noseCenterX, noseCenterY,
    noseCenterX-60, noseCenterY);
  line(noseCenterX, noseCenterY,
    noseCenterX+60, noseCenterY);
  line(noseCenterX, noseCenterY,
    noseCenterX-50, noseCenterY+25);
  line(noseCenterX, noseCenterY,
    noseCenterX+50, noseCenterY+25);
}

void drawCatNose()
{
  ellipse(noseCenterX, noseCenterY, noseSize, noseSize);
}

void draw()
{
  background(0);
  // copy the mouse position into the nose position
  noseCenterX = mouseX;
  noseCenterY = mouseY;
  drawCatHead();
  drawCatEars();
  drawCatEyes();
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
}
```

The first thing you may notice is that this change actually made the program longer! That is okay, because each piece that you will work on at any given time is now smaller. Each function or the `draw` block is itself very small and easy to manage.

6.3 Functions and Code Execution Order

Something that often trips up people here is the concept that Processing jumps around while running your code. Previously, it always ran top to bottom within your `draw` block. Now, when it sees your new command, it pauses the `draw` block, jumps to your new command, does all of that, then jumps back. Within the `draw` block you still read it top to bottom, but you need to understand that it jumps out every time a command is run. The same actually happens, e.g., when the command `background` is run, but since we didn't write that command, we don't think about it.

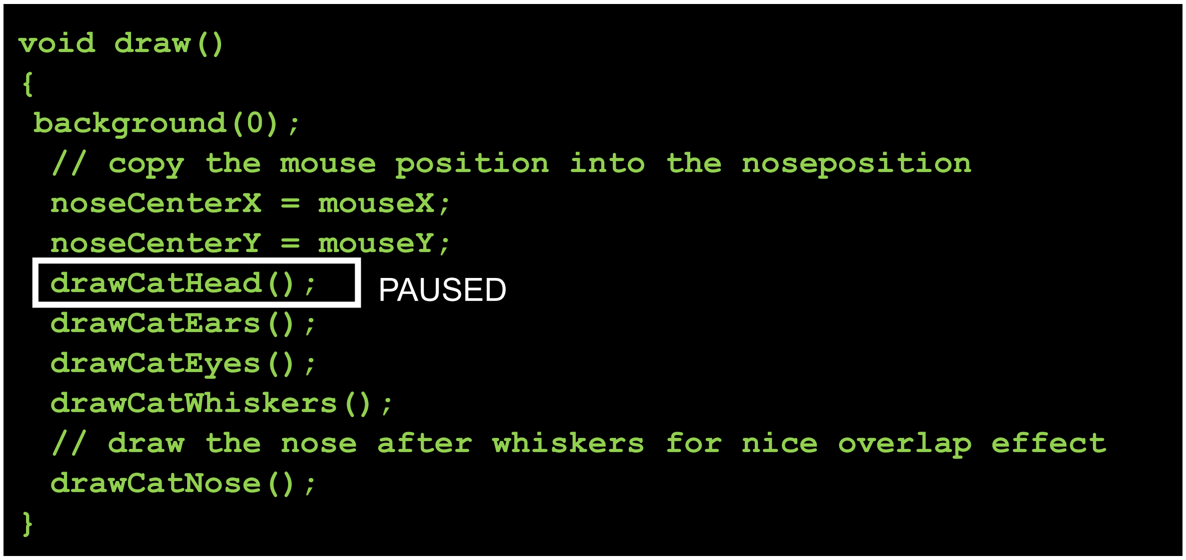
The flow of the program is exactly unchanged from what we learned in the previous Unit. First, Processing creates and initializes the global variables. Second, it runs the `setup` block. Third, it runs the `draw` block. ***The change now, is that the draw block itself can run your new commands***. What exactly happens when your command is run?

- 1) draw block is paused
- 2) your command is completely run, top to bottom
- 3) draw block is resumed where it left off.

Let's look at the cat face draw block.

```
void draw()
{
  background(0);
  // copy the mouse position into the nose position
  noseCenterX = mouseX;
  noseCenterY = mouseY;
  drawCatHead();
  drawCatEars();
  drawCatEyes();
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
}
```

When this starts, everything happens as expected. The background is cleared to black, the comment is ignored, and the mouse position is copied into the nose position. However, when the `drawCatHead()` command is encountered, the draw block pauses where it is



```
void draw()
{
 background(0);
  // copy the mouse position into the noseposition
  noseCenterX = mouseX;
  noseCenterY = mouseY;
  drawCatHead();
  drawCatEars();
  drawCatEyes();
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
}
```

And jumps off to call the `drawCatHead` code

```
void drawCatHead()
{
  ellipse(headCenterX, headCenterY, 300, 300);
}
```

When this is complete, the draw block un-pauses, and continues where it left off. Then it encounters the `drawCatEars` command, pauses again, jumps to execute the code, etc.

Important: ***You should start developing your ability to trace through code this way, to understand what happens at what spot, what pauses, where it jumps to, etc. This will be crucial to completing your assignments and exams as the course progresses***.

6.4 Functions Calling Other Functions

I said earlier that you can call your user-defined functions from anywhere that you would use any other command. Until now, that meant inside the `setup` and `draw` blocks only. Now that you have other blocks with your own functions, you may wonder if you can call other commands within those. Well, you definitely can! In fact, we could imagine that our cat face draw block could be simplified even further. We can take all the commands that draw the components of the cat face, and wrap them in another function called `drawCatFace` as follows:

```
void drawCatFace()
{
  drawCatHead();
  drawCatEars();
  drawCatEyes();
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
}
```

Then, the draw block is simplified even further:

```
void draw()
{
  background(0);
  // copy the mouse position into the nose position
  noseCenterX = mouseX;
  noseCenterY = mouseY;
  drawCatFace();
}
```

This makes sense, for example, if you were going to draw a lot of animals. Now the draw block is clean and simple and leaves room for more code.

In this case, the pausing and jumping works as you may expect, it cascades.

- `draw` block runs
- `draw` block pauses at `drawCatFace`, and jumps to the function
- `drawCatFace` runs. Then pauses at `drawCatHead` and jumps to the function
- `drawCatHead` is completely executed, and jumps back
- `drawCatFace` un-pauses, then pauses again at `drawCatEars`, and jumps to the function.
- Etc...
- `draw` block un-pauses when `drawCatFace` is complete and continues.

Advanced: Can user-defined function call themselves? Or, can a call b, which then calls a, making a loop? **YES**. This is called recursion, a topic we do not cover properly in this course. Recursion is difficult, and you should avoid it in this course in general, as it is very easy to get wrong. In fact, with recursion done incorrectly, you can crash Processing and lose your work. **Save often!**

6.5 Functions and Scope

Remember from our earlier discussions that scope is the range within which a variable exists. Outside of that scope, you cannot access or work with that variable. In general, ***any time we have a block we have scoping rules***. There is no exception here with functions, and in fact, you'll find that the scoping matches the rules you learned when we started active processing.

Since functions have their own code block, then each function has its own isolated variable scope. For example, the following code does not work:

```
void drawCircle()
{
  ellipse(location,location,5,5);
}

void draw()
{
  int location = 10;
  drawCircle();
}
```

Although you create a variable called `location`, and it is set before the `drawCircle` command is called, inside the `drawCircle` function the `location` variable is out of scope – it cannot see it. Processing will not run this code, saying that it cannot find the variable. Again, this is just what we already learned with the `draw` and `setup` blocks. How do you fix this? Make the variable global.

What about having the same variable name in multiple functions? What happens then?

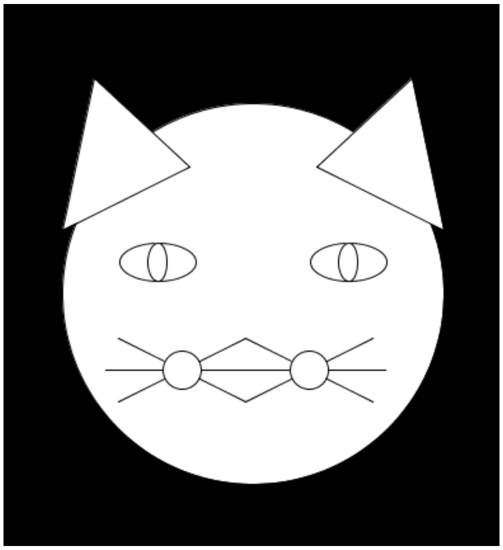
```
void drawCircle()
{
  int location = 15;
  ellipse(location,location,5,5);
}

void draw()
{
  int location = 10;
  drawCircle();
}
```

variables in different scopes can have the same name, but they do not share data – they are completely separate. Again, this is the same as we learned earlier. In this case, although they have the same name, to the computer, they are different variables.

One last thing: ***variables within a function are called local variables***

6.6 Reusing code

I promised at the beginning of this unit that user-defined functions help you to re-use code. Let's quickly look at an example of how this may work. What if we wanted to have a mutant cat – one with two noses! Since we already have the code to draw a cat nose, it would be great if we could somehow re-use that to draw a second nose. Luckily, our example is setup to help make this easy.



There are two keys to seeing how this may work. One is realizing that we can use our user-defined functions again and again. Two, is remembering that our nose code uses global variables, `noseCenterX` and `noseCenterY`, to determine where to draw the nose. So our strategy is simple:

- set some desired spacing between the noses, and save it in a global variable
- set the nose center variables to the left or right nose, and call our nose and whisker drawing functions
- change the nose center variables, and re-call the same functions

And it works! Here is my new `drawCatFace` code. Of course, the draw block no longer copies the mouse location into the nose variables.

```
void drawCatFace()
{
  drawCatHead();
  drawCatEars();
  drawCatEyes();

  // draw nose and whiskers 1
  noseCenterX = mouseX - NOSE_SPACING/2;
  noseCenterY = mouseY;
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();

  // draw nose and whiskers 2
  noseCenterX = mouseX + NOSE_SPACING/2;
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
}
```

By putting that code into our user-defined functions, we were able to re-use it very simply. Now, if we find a bug in the nose or whisker code, and we fix it, it's fixed for both noses! We don't have to change the code in just one spot. As an exercise, it may be interesting to go back and modify the original cat face without functions, to have two noses, to see how messy it becomes.

6.7 Top Down Programming

Perhaps the biggest advantage of being able to make you own functions is the ability to break your program into smaller pieces as you see fit, to simplify your problem. This is perhaps the single most useful thing you will use in programming, the ability to take ridiculously hard problems (like, make the newest flashy video game) and organize it so that you can focus on one small piece at a time (like, draw a cat eye or move a nose). This approach to programing will continue throughout the degree, through Object Oriented Programming, data structures, algorithms, and so forth – basically, we (as humans!) are not that smart and need to develop better tools and techniques to make us smarter.

Now that you can make your own commands, you can learn a common technique for helping to simplify your programming job, called top-down programming. Simply put, top-down programming is where you first look at the overall problem to be solved, and slowly break it down into smaller and smaller chunks until you have something you can code up. For example, a top-down approach to the cat face example is to first define the high level structure: the cat has a face, eyes, ears, and a nose. Then, we start to think about how to do each of those parts. For example, a nose has whiskers and a circle. Finally, we write the code to make those parts.

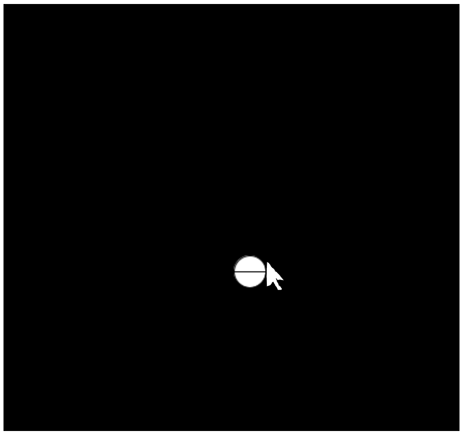
Alternatively, a bottom-up approach is the opposite: we start by building robust pieces that do specific jobs, and then connect them together to solve problems. Lego is generally done bottom-up – you start with simple blocks and start building it into something, without necessarily having an overall plan.

In the real world, there are benefits and problems associated with both styles. You will find yourself going back and forth depending where you are in a project, or which actual problem you are solving.

For this course, y***ou learn top-down programming as a means of helping you take a problem that may be somewhat overwhelming, and break it down into chunks until those chunks are manageable***.

You will get a lot of practice with this throughout the course, and I strongly recommend you use this approach when you feel overwhelmed. We will go through a toy example with somewhat simple code for illustration purposes.

We will make a program that has a planet moving around the screen. It will do a few things: move toward the mouse, with some friction to slow it down, and move in a repeating horizontal orbit from left to right. The planet itself is a circle with a line through it. There is a lot happening here, so it can be helpful to use top-down programming to simplify the problem. The general approach to the problem will be to calculate the desired move, do the move, and draw it.



We will approach this with four steps:

1. Write the program as a series of steps in comments, in English.
2. Turn each step into a function name (command)
3. Create empty functions
4. Start implementing the functions

(Step 1) Using top-down programming, first we plan the steps of the program in English and make sure it makes sense on that level. Here is an example of what our English version may look like.

- Clear the background to black
- Calculate the attraction to the mouse
- Add friction
- Update the orbit
- Move the planet based on the calculated move
- Draw the planet.

Whew! Did I miss anything? **Point:** We are thinking through our program without doing any computer programming at all. We are just planning things out and listing the tasks that need to happen. The next step is to toss all of this list into our Processing program (as comments!) and convert them to function calls.

(Step 2) Wouldn't it be nice if we could simply convert this into commands and have it work? Imagine that you could simply take our above list and generate:

```
void draw()
{
  background(0);
  attractToMouse();
  addFriction();
  updateOrbit();
  movePlanet();
  drawPlanet();
}
```

In fact, this is what user defined functions are all about. We can think about the program abstractly, from a high level. We determine what needs to be done in English first (often in comments). We turn the comments into new commands. We then create those commands: for each command, we only focus on a narrow problem. For example, in `attractToMouse()` we don't worry about drawing or the orbit, just the calculation to move the planet to the mouse.

(Step 3) We next just implement empty functions, to make the skeleton of the program. Once we are done this, the program should run – but nothing will happen.

```
void attractToMouse()
{
}

void addFriction()
{
}

void updateOrbit()
{
}

void movePlanet()
{
}

void drawPlanet()
{
}
```

Just to re-cap, by this point, we have invented a bunch of functions that we think will help us do our job, and wrote up those functions as empty stubs. We haven't done any actual problem solving yet.

(Step 4) Start implementing the functions. The nice thing about this approach is that you can, for the most part, think about one problem at a time. Also, while you are implementing a function you can develop the required global variables and constants.

Important: your initial functions may actually be a bad idea, and may not work. Don't be averse to re-factoring, that is, re-organizing your solution as you see problems with it or see a better approach.

First, let's do the easy one. Drawing the planet. We need to have global variables for the planet position, since it will move around and we need to remember where it was last time. We also need a global constant to specify the planet size:

```
// main planet
final int PLANET_SIZE = 30;
int planetX = 0;
int planetY = 0;
```

With these globals, then we can draw the planet. The nice thing here is that we can use the planet location variables and the size, and solve the basic geometry without worrying about the other tasks. Here is my draw code, but try implementing by yourself first.

```
void drawPlanet()
{
  ellipse(planetX, planetY, PLANET_SIZE, PLANET_SIZE);
  line(planetX-PLANET_SIZE/2, planetY,
       planetX+PLANET_SIZE/2, planetY);
}
```

Next, let's attract to the mouse. All that this code does, is to determine what the move would be to move the ball to the mouse. This should be a simple subtraction calculation. However, we need new globals to store our guessed move. This is better than modifying the planet position directly, since then we can scale our move (e.g., for friction!) by dividing it, before actually performing the move.

```
// keep track of our calculated move
int planetMoveX = 0;
int planetMoveY = 0;
```

To do this, we just calculate how far the ball is from the mouse, and add it to the planned move. As the risk of being repetitive, try to solve this problem only thinking about the task at hand, and let yourself forget about the other tasks (like the orbit or drawing). This is one of the key benefits of using user defined functions.

```
void attractToMouse()
{
  int diffX = mouseX - planetX;
  int diffY = mouseY - planetY;
  planetMoveX = planetMoveX + diffX;
  planetMoveY = planetMoveY + diffY;
}
```

The next function in our list is to add friction to our calculation. There are a lot of ways to add friction (some physically correct, and some not ☺), but what we will do is divide our planned move by some amount. This way, larger moves get cut more, and smaller moves less. We need a global to store the friction amount

```
final int FRICTION_DIV = 10;
```

And to add the friction, we just divide the intended movement by our friction amount.

```
void addFriction()
{
   planetMoveX = planetMoveX / FRICTION_DIV;
   planetMoveY = planetMoveY / FRICTION_DIV;
}
```

Now is a good time to do the move code. If we assume that the move variables are correctly done, then we can just add the planned move to the planet location and we're done.

```
void movePlanet()
{
  planetX = planetX + planetMoveX;
  planetY = planetY + planetMoveY;
}
```

Now you should have a program that runs quite well, and a planet that approaches the mouse as you move it around. To continue the program, you need to solve the orbit problem. In this case, I recommend using a fixed orbit size, and using modulo to make it loop when it gets too large. Try finishing this exercise on your own time. Note: you may have to modify your drawing code to get this to work!

Again, to re-iterate the benefits of this approach, you have so far solved most of the program without thinking about the orbit at all. This method enables you to focus in on one small piece at a time, and solve the whole problem part by part. Top-down programming:

1. Helps you focus on smaller problems and not be overwhelmed
2. Helps make self-commenting code, via descriptive function names
3. Helps you make a larger plan as you go

Check your Understanding

6.8 Check your Understanding: Exercises

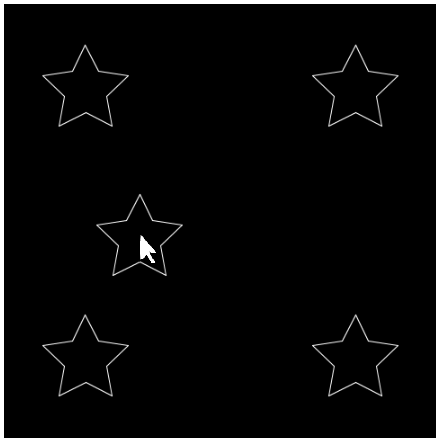
Exercise 1. Create a space ship shooting at an enemy (unfortunately, not animated). Here is the draw and setup blocks.

```
int shipX = 0;
int shipY = 0;
void setup()
{
  size(500,500);
}

void draw()
{
  background(0);
  shipX = mouseX;
  shipY = mouseY;
  drawSpaceship();
  drawEnemy(); // above spaceship
  drawMissile(); // between ship and enemy
}
```

a. Implement the user-defined functions specified with your own solutions.
b. Can you make the bullet animate, moving from the spaceship to the enemy, and repeating?

Exercise 2. The following program draws several “stars”. You can decide yourself how to draw a star, but using two triangles (one upside own) is a good way to do it.

a. Make two global variables: `starX` an `starY` that specify where to draw a star.
b. Make a user-defined function that draws a star at the globally-defined location
c. In you draw loop, draw several stars by alternately changing `starX` and `starY`, and calling your function. Also, draw one under the mouse.



Exercise 3. You will make a program that lets you draw fairly generic faces. You will have functions to draw eyes, nose, mouth, and head, which use global variables to determine their properties. Then you make other functions that modify the globals and call your helper functions to make different faces. It is up to you whether you take a bottom-up or top-down approach on this one.

a. Make global variables to specify all the parameters, e.g., head width and height, eye width and height, pupil ratio, etc. You have some flexibility here. Also, make variables for each item to specify *where* to draw them.
b. Make user-defined functions that draw the parts using the global variables.
c. Make two user-defined functions: `drawNormalFace()` and `drawSillyFace()`. These set the proper global variables and use your other user-defined functions to draw two different faces at two different locations on the screen.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

UNIT 7. REAL NUMBERS – FLOATING POINT

Summary

Up until now in the course, all your calculations have involved whole numbers. We did not have any mechanism for using numbers with decimals, known as *real numbers.* Here we learn how to use a new data type to enable us to use decimals.

In this section, you will…

- Learn how to create variables that can store real numbers
- Learn how to work with real numbers, e.g., basic math functions
- Revisit high-school trigonometry, and learn about sine, cosine, and tangent in processing
- See how we can make random numbers
- Learn new basic math shortcuts for real numbers and integers
- Learn a few new globals to get the canvas size
- See the limitations of real numbers when implemented in computers

Learning Objectives

After finishing this unit, you will be able to …

- Create variables to store real numbers
- Work with real numbers, e.g., do basic math operations
- Do basic trigonometry to help with graphical calculations
- Use shortcuts for addition, subtraction, multiplication, division, and modulo
- Get the canvas size from new global variables

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

7.1 Introduction

We learned already about integers in Processing, but sometimes we cannot be limited to whole numbers. Real numbers are basically the same as integers except you can have a decimal portion attached to them. With real numbers, the operation 5/2 gives us 2.5, not 2 remainder 1. This may seem more natural to you, but you will still end up using integers an awful lot. There are several reasons, but the primary one is that real numbers cannot be fully trusted (more on that later!), while integers can. Also, computers are generally faster when working with integers, which is important in a lot of applications.

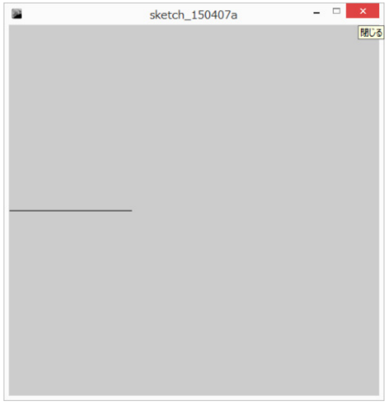
To use real numbers in Processing we need a new data type: floating point numbers. This works basically the same as integers, except we use the word `float` instead of `int` to create the variable, and now we can assign decimal portions to the variable:

```
float variableName;
```

Let's try it with the following static program:

```
size(500,500);
float percent = 0.33;
line(0,250,percent*500,250); // draw percent way across
```

yes it works! By using a floating point, we can now calculate things like drawing 33% across the screen. This scales nicely, and you can use real numbers throughout your program.



Why do we call them floating points, and not just real numbers? This is just because of how they are stored inside the computer. The computer uses something akin to scientific notation to store these numbers. For example, 1234.5 could be stored as $1.2345\text{x}10^3$. Just by changing the exponent, we can go from this large number to a small one, that is, $1.2345\text{x}10^{-3}$ gives us 0.0012345. The point (dot) can float around, giving us very large and very small numbers.

7.2 Real Numbers and Calculations

For the most part, you use the same basic calculation syntax we already learned for addition, subtraction, multiplication, and division. However, there is a big gotcha: let's try to shift the above example slightly:

```
float percent = 33/100;
line(0,250,percent*500,250); // draw percent way across
```

What happens? Try running this program. The line disappears! Let's investigate by using print to output the percent:

```
println(percent);
```

you get 0. What happened? Why would 33/100 give you zero?

This happens because Processing is still doing integer division. Let's look at the line that is causing the problem.

```
float percent = 33/100;
```

You would think that Processing can look at this, see that you are storing into a `float`, and do what you want. Unfortunately, computers are stupid. Processing is very very near sighted. When doing that division, it only looks at the numbers right beside it, and not the bigger context. When it sees 33/100, it thinks: oh! 33 is an integer. 100 is an integer. Let's do integer division. ***If the numbers in an operation are integers, integer arithmetic is used. In technical terms, if the operands are integers, Processing uses an integer operator.***

How do we fix this problem? We can basically re-think the above statement about integers. ***If either of the numbers in an operation are floating point, Processing does floating point math.*** So we can fix this by making one of the numbers floating point. How would we do this? How about

```
float percent = 33.0/100;
```

Yes! It works! Processing looks at the operation, sees a floating point, and uses floating point math. The 100 could also have been changed to 100.0, or both numbers can change. You get the same result.

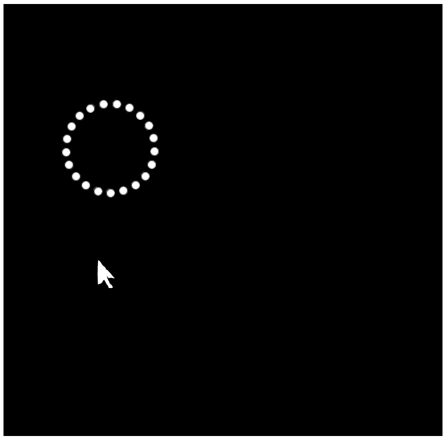
Unfortunately, this can get very tricky. Let's think about the following example:

```
float result = 1.0 + 1/2*3.0; // expected 2.5
```

If you test the output, you get 1, not 2.5! Integer division is at the root of the issue here. To figure this out, let's look at the order of operations. Multiplication and division happen first, going left to right. So, the first calculation we see is 1/2. Since Processing is so narrow sighted, it only sees integer / integer, and does integer division. 1/2 gives you 0, so we get 1.0 + 0*3, = 1.

Now, let's try an example using floating points. Let's make a bubble go in a circle around the mouse, like a planet orbiting it.

Unfortunately, we cannot so easily do that with the current tools that we have. We first need to learn how to do trigonometry in Processing – we need sine and cosine to calculate the points on the circle.



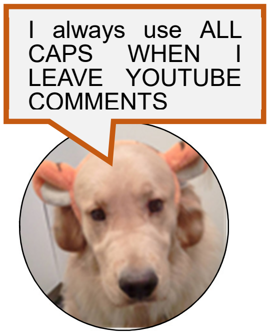
7.3 **Trigonometry in Processing**

When you left high school, I just know one of your dreams in university was to get more use out of sine, cosine, and tangent – those amazing geometric functions that are so useful! Well here's your chance! Trigonometry is extremely useful and powerful in computer graphics, and we'll use it in this course.

First, I have to point out that Processing uses radians, not degrees, for angles. Remember that radians go from 0 to 2*PI whereas degrees go from 0 to 360. There are functions in processing to go between radians and degrees, but I'm not going to teach them and I recommend you just stick to radians. Besides, doing the conversion yourself is really easy.

The first thing to learn is that Processing has a built in global *constant* called `PI`. It is a constant because you are not allowed to change its value (good!!). You can use this constant pretty much just like any other variable, just remember to use ALL CAPS – as we learned earlier, this is a convention for letting people know the variable is constant:

```
println(PI);
```



In addition to this constant, processing provides the following commands. I am going to present these commands in a new way, a way that is common in programming. When you describe a command, it is useful to specify what kind of data you give it (parameters) and what kind of data it gives you, like this:

```
type commandName(type parameter);
```

the first type explains what kind of data the command gives you, and the parameters are what you give it. Let's try this for the trigonometry functions

```
float sin(float radians); // sine of radians
float cos(float radians); // cosine of radians
float tan(float radians); // tangent
```

The sine command (sin) takes radians as a float, and gives you the sine result as a float. You can use the result to store into a variable or print out to screen, e.g.,

```
print(sin(PI));
```

or store the result in a variable:

```
float result = sin(0.4);
```

And you can use this in your calculations.

You also have the inverse commands. These are often tossed into text books as sin^{-1}, cos^{-1}, tan^{-1}, but are also called arcsine, arccos, and arctan. These go backward, and given a number, gives you the ratio (e.g., opposite / hypotenuse) that created that ratio. You won't use these as often, but it's useful to know that they exist.

```
float asin(float ratio); // inverse sin
float acos(float ratio); // inverse cos
float atan(float ratio); // inverse tan
```

Advanced: atan has a particular problem. Remember that tan(θ) is opp / adj. Depending on which quadrant you are in, opp and adj can be both negative, positive, or a mix. The problem is that if both are negative, you get the same ratio as if they were both positive, since the negative signs cancel out. The only way to deduce which quadrant the angle belongs in is to know those original plus and minus signs, and do a case by case calculation. Luckily, processing helps you out with this with the atan2 command that does the calculation for you.

```
float atan2(opposite, adjacent); // OR easier to understand:
float atan2(y,x); // gives correct angle from a y and x offset
```

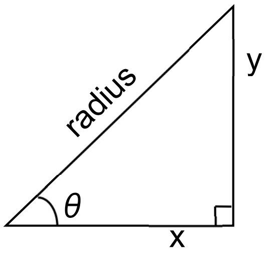
Now, with these functions, we can make a circle spin around the mouse cursor! This is how we do it. As with other examples, we approach this by first solving the static case: that is, place a ball at some angle and distance from the mouse, but don't animate it yet. Let's pick some angle and call it theta (θ).

```
float theta = 0; // angle of 0 radians, straight right
```

We also need to specify how far from the mouse cursor our ball should orbit. This is our radius.

```
float radius = 50; // distance from the mouse to orbit
```

Given our angle theta, and radius, where should we draw the ball? Luckily, this is basic trigonometry! As shown in the diagram, we ***if we are given the angle and the radius, we can easily calculate x and y:***



- sin(theta) = y/radius → sin(theta)*radius = y
- cos(theta) = x/radius → cos(theta)*radius = x

It is a good idea to review your basic trigonometry triangle as shown in the diagram, since this will come up again and again.

Let's translate this into processing:

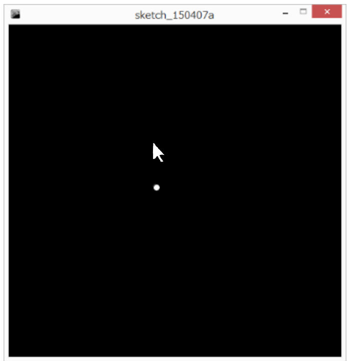
```
float x = cos(theta)*radius;
float y = sin(theta)*radius;
```

that's it! We have calculated the x and y location at the given angle and radius. Let's test it by drawing an ellipse at our new coordinate.

```
ellipse(x,y,10,10);
```

It works! You should see a ball to the right of the origin (top left of the screen). Now let's make it relative to the mouse cursor. All that we have to do is to add the mouse location to it:

```
ellipse(mouseX+x, mouseY+y, 10, 10);
```



As you can see, in processing, 0 radians (or degrees) is straight right along the x axis, as you would expect. Unlike you learned in high school, however, angles increase clockwise in Processing, because the y axis is upside down! Test this by looking at where the ball goes at PI/2 (45 degrees), PI, etc.

So now that we can draw a ball at a given angle relative to the mouse, how do we make the ball rotate around the mouse? We can do this the same way that we did for the bubbles: we increase the ball angle by some delta each time it is drawn. Let's make a new variable called `delta` that determines how fast the ball moves. That is, how much to change the angle each frame.

```
float delta = 0.1;
```

and, after drawing the ellipse, increase your theta by this delta:

```
theta = theta + delta;
```

Every time you draw, theta will increase. Now, since this is radians, we can just let it get bigger – once it gets bigger than 2 PI it will automatically loop around. That is, 3PI radians gives you the same angle as PI radians.

Can you get it to work with all these pieces? Here is my final code:

```
float theta = 0;
float delta = 0.1;
float radius = 50;
float size = 10;

void setup()
{
  size(500, 500);
}
void draw()
{
  background(0);
  float x = cos(theta)*radius;
  float y = sin(theta)*radius;
  ellipse(mouseX+x, mouseY+y, size, size);
  theta = theta + delta;
}
```

I really need to emphasize the importance of reviewing your basic trigonometry. This course is not about trigonometry, but you are expected to know your basic math to do work. If you are still not up to speed on this, you will get bogged down in examples later, and will find yourself missing what you are really supposed to be learning.

7.4 **Random**

Let's make a program that continuously draws lines from the mouse cursor to random spots on the screen. Until now, everything we do is static, but Processing provides a nice simple way to generate a random number:

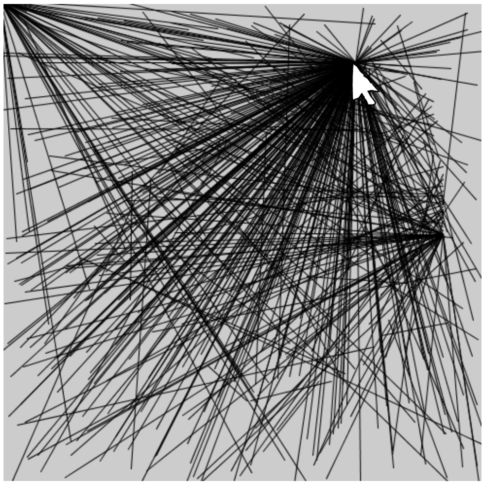
```
float random(float high)
```

This generates a random number `[0..high]`. That is, from 0 up to high, but not including high. For example,

```
random(500) // number from 0..499.9999
```

This gives you a number that you can store in a variable or use in a command. Fist, try drawing a line from the center of the screen to a random point in a static program. This assumes a canvas of 500x500 pixels.

```
line(250,250,random(500),random(500));
```



run this program, and you will see a random line. Run it again. And again... and you will get a different line each time. Let's make this active, but instead draw from the mouse location.

Random is pretty easy to use. The complications come when you want to generate specific ranges of numbers, as we often get off-by-one errors since the math is easy but tricky. The other complication is that this is a floating point, and we often want integers (e.g., for a dice roll). We'll learn how to fix that in a later chapter.

7.5 Limitations of floating point numbers

In the abstract mathematical world, real numbers have infinite possibilities. A number like 1/3, or 0.6666 repeating, goes on forever. Other numbers, like pi, are irrational and never repeat (and go on forever). Other numbers, like 0.1, is a lot better behaved, and easy to represent.

Unlike the abstract mathematical world, the real world has limitations. Computers have limited memory. When storing 0.6666 repeating, or pi, a computer cannot store infinite digits because they don't have infinite memory. By necessity, computers store approximations of real numbers. How closely the number is approximated depends on how much memory it uses – using more memory lets you approximate better, etc.

So, when a computer stores a number like 0.66666, it will only store so many sixes. Try the following:

```
println(2/3.0);
```

You should get the output 0.6666667, which is a reasonable approximation.

This same applies with larger numbers, not only the decimals. For example, try the following:

```
println(2/3.0*10000);
```

The output is 6666.667 – by making the number larger, we lose some of the decimals. That is because we still have the same amount of memory for the approximation, some of it is now just used for the whole number portion. If you want, you can even move the digits completely out of the decimal portion:

```
println(2/3.0*10000000);
```

Which gives you 6666667.0! You can take this further, but no matter how large or how small your number gets, the computer has limited precision with which it approximates the true number.

If you remember, floating point numbers are stored as scientific notation. As such, your range of possible values is very large – just by changing an exponent, you can get very very large and very very small numbers. However, no matter how large or small, you still only have so many digits of precision. For example, if you wanted to store 2/3.0 multiplied by a billion, you would get something like 6,666,660,000. A large number, but not entirely accurate down to the number level. In Processing, and in Java, the range of possible values for a float is roughly from 3.4x10^{-38} to 3.4x10^{38}, and can be negative or positive, although the precision is limited. You don't really need to be aware of this range except in special circumstances.

There is another important limitation of floating point numbers. Because we think in base 10, and a computer thinks in base 2 (1s and 0s), not every number can be stored perfectly. This is a really hard concept to get your head around, so don't worry about the detail for now (but see the advanced blurb below).

For most people, this situation is just fine. For the calculations you will do in this class, these limitations will not usually pose a problem. However, you must be aware of the limitations, as they explain certain things that you will see as you go through programming. Here is an example to illustrate: do the following two lines of code give me the same output? Think about it:

```
println(0.7);
println(0.6+0.1);
```

Mathematically, the results should be the same. However, try running the code:

```
0.7
0.70000005
```

The result is different! Because of how the computer stores the numbers, you can't trust the result. At this point, you don't need to worry about exactly why this is but just remember: ***floating point numbers cannot be trusted to be exact.*** This will keep coming up again in the course, but if you need exact numbers (such as for money!!) you should not use floating point variables.

Numbers cannot be trusted? I knew it! Never trust statistics!

Advanced: Here is a way to think about this limitation. If you have an 8 bit number, we have 2^8 combinations, and it can store 0...255. Even though it goes form the smallest binary number (00000000) to the largest (11111111), when we move to decimal, we only hit a small part of the 3-digit range (000-999). If we were to take this same number and upgrade it to floating point, e.g., with scientific notation, then add an exponent. Say the exponent can be small or large, then we can now take a number like 25 and have very large numbers, say, $2.5x10^5$ or $2.5x10^{-5}$. We already saw that we have limited precision, but now, how could we store the number 9.99? The 8 bit underlying number only goes from 0...255, so we cannot even do 2.56. Computers solve this by storing the number slightly differently, in a somewhat complex way. The result is that we get the full range nicely in decimal (000-999), but we lose some precision about which numbers can be stored exactly. You'll learn this in greater detail in a later course.

7.6 Dusty Corners: Shortcuts and new Globals

There are a few extra things to show you at this point. These did not come up earlier just for simplicity's sake.

First, Programmers hate typing, and will find any opportunity to shorten what they have to type. There is a whole range of simple shortcuts that you will see and find yourself using, I introduce a few of those here.

It is very common – as you have seen in our examples – to do operations of the following pattern:

```
variable = variable + number;
variable = variable * number;
```

and so on. These have short hand versions:

```
variable += number; // variable = variable + number
variable -= number; // variable = variable - number
variable *= number; // variable = variable * number
variable /= number; // variable = variable / number
variable %= number; // variable = variable % number
```

These safe just a few keystrokes, but you will find yourself using them!

Another shortcut is incrementors and decrementors. Even more common than the examples above is to increase or decrease a variable by 1. We saw this often in our exercises. For example:

```
variable = variable + 1;
```

we already learned that we can shorten this to

```
variable += 1;
```

but there's even a quicker way!

```
variable++; // variable = variable + 1
```

There is also

```
variable--; // variable = variable - 1
```

Again, it may seem silly to have such shortcuts, but you will find yourself using them.

That's why it's called C++! It's the next version of the C programming language

Finally, there are two new globals for you to use. Once the canvas size is set, you can read from two global variables what the current size is. You have `width` and `height`, which tell you how many pixels there are. For example, you can draw a line from the top left to the bottom right as follows

```
line(0,0,width-1,height-1);
```

Why the minus 1? Maybe review that from unit 2?

7.7 Example: block moving around randomly

Let's make a square block that moves randomly around the screen. Moving in random directions around the screen is not the same as moving to random locations! If a block is at a certain location, then it should move in some direction away from where it is.

First, make some globals to store the block position (you need two for this) and size. You will also need a variable to store how quickly it can move. Make the block speed around 5 to start. Make these floating point variables. In the draw block, clear the screen, leave a blank for our moving code, and draw the block using the rectangle command.

The first challenge is thinking about how to move the block randomly. An easy way to do this is to move the X and Y separately. Start with just X for now, to simplify the code. Make a variable called `moveX`, and use the random command to generate how much that the block should move, using your global variable. For example:

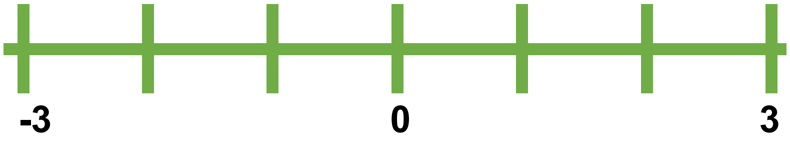
```
float moveX = random(MOVE_MAX);
```

Then, add your `moveX` to your block position, so that the block *moves* by this amount in the frame.

```
blockX += moveX;
```

Try this out. You should notice that the block moves by a random amount – however, there is a caveat! The block only moves to the right! If you think about this, it makes sense: random gives us a positive number, and we add that to the `blockX`, so the `blockX` can only get bigger. This means it can only move to the right.

You need to do some clever but basic arithmetic to get this to work. Basically, we want the block to move either in the negative direction by our maximum, or in the positive by our maximum, or no movement at all. So, our actual range of potential movement is double the maximum. Consider the following number line, with a maximum movement of 3.



We want a random number somewhere on this line. A call to `random(3)` gives us a range of almost 4, from 0...3.999. If we use `random(2*3)`, double the maximum, we get a range of almost 6, 0...5.999. If you look at the number line above, you can see that this is right – we need a range of 6. Now, the result from random needs to be shifted to match our number line. If we subtract three: `random(6)-3` gives us a range from (0…5.999)-3, which is -3…2.999, which matches our number line. To turn this into code, we need a new calculation for `moveX`:

```
float moveX = random(MOVE_MAX*2)-MOVE_MAX;
```

Now, you can adjust this to also move in the Y direction.

Finally, there remains a problem: the block moves off of the edge of the screen. You can fix this in your own time, using the min and max functions. Review the earlier section if you are not sure how this may work. (Note: `min` and `max` work just fine with floats).

Check your Understanding

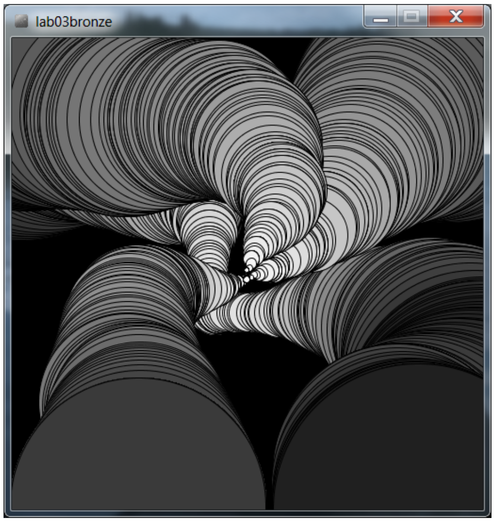
7.8 Check Your Understanding: Exercises

Exercise 1. Make a program that converts from Celsius to Fahrenheit. Make a static program for this, and use floats for all your variables. Make one variable to store a Celsius value (e.g., 25), and then convert it to Fahrenheit using the following formula: f = 9/5*c + 32. Output your result using `println`. Be careful of integer division! 20c should give you 68f.

Exercise 2. In section 7.3, you made a ball orbit the mouse.

a. Make the ball orbit in an ellipse instead of a circle. You can do this by using a different radius for the x and for the y calculations.
b. Make multiple balls orbit the mouse at different speeds. E.g., make three.
c. Make one move in the opposite direction (counter-clockwise).

Exercise 3. You will make an interesting drawing program that lets you make images as shown in the inset. To get this working correctly you will need to create an active processing program, and use floating point numbers.



a. Create an active processing program and in the setup block create a canvas window with a black background. Your program should work with a window of any size, square or rectangular.
b. You will be drawing a circle (i.e., an ellipse) at the mouse position. However, the size and fill color of the ellipse will be calculated based on the distance from the mouse to the center of the window, as explained below.
c. The following formula can be used to calculate the distance between any two points (x_1, y_1) and (x_2, y_2): $\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. To get the square root, use the built-in function `sqrt(x)` which returns $\sqrt{x}$. It would also be convenient to use the function

`sq(x)` which returns x^2. Both work with `float` values.

d. Draw a new circle, every time, in the draw block, with its center at the mouse position, and a *diameter* equal to the distance from the mouse to the center of the window. Test it. You should get an image like the one above, except that all the circles will be white

e. Now change the fill color of the circle. It should also depend on the distance from the mouse to the center of the window, but it should be 255 (white) at the exact center (distance 0), and 0 (black) if the mouse is as far away as possible (the distance from (0,0) to the window's center):

   i. Calculate the maximum possible distance.

   ii. Convert the mouse's distance to a proportion of this distance (a float value from 0.0 to 1.0).

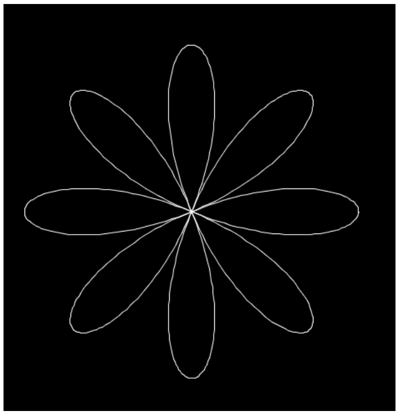
   iii. Use this to compute the correct color. (You want 255 if the proportion is 0.0, and you want 0 if the proportion is 1.0.) The formula is very simple. Think about it.

Exercise 4. Make example 7.7 also move toward the mouse, in addition to moving randomly. This will simulate what an enemy in a video game may do – move about, by tend toward your player.

a. First, calculate how far away the block is from the mouse. This is a very simple calculation if you do x and y separately. Think about it: how far in the x direction? How far in the y direction?

b. If you add this result to the block position, it will just be *at* the mouse. Instead, make it move *toward* the mouse; scale your results down by dividing by some number.

Exercise 5. You will implement a mathematical function which generates a class of shape called a *rose* in mathematics. There are details of this online (http://en.wikipedia.org/wiki/Rose_(mathematics)) but basically this is like a Spirograph. It creates images such as the one shown on the right.



You can generate roses with different numbers of petals, using a *parameterized function*. Some *parameter* variable (often *t*) is allowed to vary smoothly from one value to another (often from 0 to *2π*), and then the quantity of interest (here we want an `(x,y)` point) is obtained using a formula that depends on *t*. There are usually some other constants as well.

To get a “rose” use:

$$\mathrm{x} = \cos(kt)\cos(t) * r + x_c$$

$$\mathrm{y} = \cos(kt)\sin(t) * r + y_c$$

The value *k* affects how many petals are drawn. Try 4, which would give the shape above. The value *r* is the radius of the rose (for us, that will be in pixels). Use most of the window, as shown. The point (x_c, y_c) is the centre of the rose (which should be the center of the window). As *t* changes from 0 to 2π, in many small steps, the points `(x,y)` generated by the formulae above will trace out the rose.

a. Use a global variable *t*, which starts at 0, and increases by some small amount every frame. (Here, a single-letter name is OK, since it’s traditional. But don’t make a habit of it!) Use a global constant for the amount of change each frame. A value of 0.01 will give a smooth curve, but 0.05 is faster.
b. Also use global constants for the other values needed by the formulae (*k* and *r*). Always use the center of the window for (x_c, y_c).
c. Each frame, calculate a new point `(x,y)`, and draw a line from the previous `(x,y)` point to the new one. You will have to keep track of the previous point for yourself, using global variables. This gets a little tricky, but you won’t see why until you see the result!
d. Make sure that *t* never gets larger than two pi.
e. You can get cool variants by changing the formula a bit. Replace $\cos(kt)$ with $(\cos(kt) + a)$ (the brackets are important). Then set *a* to some value like 0.1, 0.5, and see the difference.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 8. Text in Processing: Basic Strings

Summary

Up until now in the course, all of your computation and variables have been using numerical values. Here, we learn how to use text in processing.

In this section, you will…

- Learn how text is stored in Processing
- See ways to work with text, such as combining multiple segments into one, and converting back and forth to numbers
- Learn how to put text to the console, and on the canvas.
- Learn about the difference between a chunk of text and a single character.
- See how to get the length of text, and how to extract specific characters

Learning Objectives

After finishing this unit, you will be able to …

- Create variables to store text data, called Strings
- Combine multiple Strings together
- Convert between Strings and numerical data
- Get the length of a String, as an integer
- Create a character variable
- Get a specific character out of a string, from a specific index.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

8.1 Introduction

Until now in the course we have only been dealing with numerical data: integer and floating point numbers. It is time to learn how we can work with text in Processing.

In computing, a piece of text is referred to as a ***string of characters***, or a string for short. To write a string (a piece of text) in processing, you encapsulate it in double quotes as follows:

```
println("hello world!");
```

Make sure to use double quotes `"` (the shift plus single quote) not the single quote ` or ', or two single quotes `''`. Processing takes everything from the first quote to the last quote as verbatim text. If you try to put text without the quotes, Processing tries to parse the text as programming commands and you'll get an error.

While strings give you a lot of power (text is very important), they also introduce a lot of problems. A lot of this stems from the fact that strings are a new kind of data. Until now, all our data types are *primitive types*, those that just store one simple piece of data. Strings are more complicated: they can have varying lengths, they can be short or long, etc. Because of this, strings fall into a different category of data called ***Objects. Objects are ways to encapsulate more complex kinds of data in easy to use ways.*** Unfortunately, you often use Objects differently than regular primitives, and even worse, different Objects are often used very differently. In this course, we won't learn Objects, but be aware of this as the reason why Strings can be so different from time to time, and be assured that it gets easier as you learn computer science (and Objects!)

The first difference is in the naming scheme. Unlike our primitive data types, Object types should start with a capital letter. In this case, to make a new string variable:

```
String variableName;
```

Other than that you can use this like your other types so far. For example, you can have combined declaration and instantiation:

```
String name = "Jim Young";
```

The variables themselves are also usable in many ways like other variables, for example, as above we can use it in our `println` statement:

```
String name = "Jim Young";
println(name);
```

Many of the operations that we learned on numerical data (such as multiplication or modulo) do not work on String data. Luckily, this makes sense: what does multiplication mean with text? But, we have some new operators that we'll see shortly.

This is a good time to introduce the ***empty string. This is the simplest string that you can come up with, as it contains no information!*** (a little Zen?) Students sometimes struggle with this, but we'll get some practice with it. You specify an empty string by putting two double quotes together like this:

```
String empty = "";
```

If you print this out, as may be expected, you get nothing ☺. You can think of this as kind of a default, or starting value, for a string.

Advanced: how do you place a quote sign in a string? If you try to do it, it doesn't work, since Processing doesn't know the difference between a quote to end the string, and a quote as part of the string. The solution is to use what is called an escape sequence, which is a command inside a string. There are many such sequences, but the one we would use here is backslash quote. For example:

```
String message = "Hi ¥"friend¥"";
```

8.2 Concatenating Strings

The word *concatenate* may sound complex, but all that it means is to stick two things together. Concatenating strings just takes two strings and combines them to make one. For example, consider how silly the following code is:

```
String firstName = "Jim";
String lastName = "Young";
String fullName = "Jim Young";
```

This is odd because if we have the first and last name, we should be able to calculate the full name without having to type it in again. We can do this using the concatenate operator. All that you do, is put a plus sign (+) between two strings as follows:

```
String firstName = "Jim";
String lastName = "Young";
String fullName = firstName + lastName;
println(fullName);
```

I recommend that you try this to test it out. What is the output? Any problems? The concatenation worked, but we got JimYoung as our output with no space between them: Processing stuck them directly together. This I simple, because you can chain the concatenation operations together. Just use more concatenation to add a space:

```
String fullName = firstName + " " + lastName;
```

Concatenation is easy and scales up very intuitively. Let’s try making a Madlibs game. Let’s take a template string and plug in words. Here is our template:

“<exclamation>! He said <adverb> as he jumped into his convertible <noun> and drove off with his <adjective> partner.”

To convert this into processing, let’s first create the words as variables. In your own case, select your own instances of words:

```
String exclamation = "smeg!";
String adverb = "happily";
String noun = "dog";
String adjective = "red";

// calculate our output text
String output = exclamation +"! He said "+adverb+
       " as he jumped into his convertible " + noun +
       " and drove off with his " + adjective + " partner.";
println(output);
```

There is also shorthand concatenation. Commonly, just like with numbers, we do the following kind of scenario to keep adding to a string:

```
String output = "";
output = output + exclamation;
output = output + "! He said";
```

and so on. In this case, we can use the += shorthand just like with numerical data:

```
output += exclamation;
output += "! He said";
```

It just saves a few keystrokes, but you’ll appreciate it in practice ☺

8.3 Graphical Text in Processing

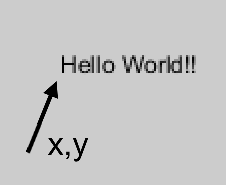
So we can toss text to the console for testing, but how do we include it in our graphical programs? We need some new commands.

```
text(string data, x, y); // draw string at x,y
```

For example,

```
text("Hello World!!", 50, 50);
```

Be careful, the x and y are the bottom left corner of the base line of the graphical text, not the top left like with a rectangle. Some text may go below this, e.g., a lowercase "g".



You can set the color of the text using the same `fill` command that you already know.

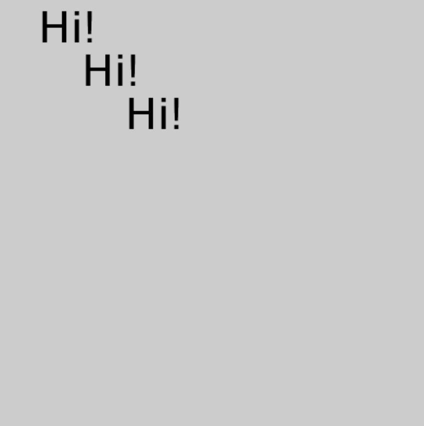
You can also set the size of the text in pixels:

```
textSize(size in pixels);
```

Make sure to set this size *before* you draw the text.

For example:

```
int size = 50;
textSize(size);
text("Hi!", size, size);
size = 100;
text("Hi!", size, size);
size = 150;
text("Hi!", size, size);
```



Processing also has other great functions, like determining how many pixels long a string will be when displayed on the canvas, or how tall it will be – these measurements can be useful for doing layouts of your interface. However, they are beyond the scope of this class. Feel free to look them up online!

8.4 Converting between text and numbers

Fundamentally, the *idea* of a number and its *textual representation* are quite different. For example, the number 5 is an abstract concept that you can store in an integer, and do mathematical operations on it. However, the textual representation of "5" is arbitrary and depends on the language. For example, these are all representation of the same number: 5 V ||||| いつつ 五 오. As another example, the number 1234.56 can be textually written as 1,234.56 or 12,34.56 and still have the same intrinsic numerical meaning and value.

As such, the string “5” is different in a computer than the integer 5. Try the following:

```
String s = "5";
int i = s;
```

Processing complains that you cannot convert from a `String` to an `int`, and it doesn’t work. Likewise:

```
int i = 5;
String s = i;
```

Doesn’t work. T***ext and numbers are fundamentally different concepts, and computers see them as fundamentally different data types***. The computer does not see them the same as you or I would.

The workaround is that ***we need specific commands to convert back and forth between numerical and string data.*** We have two commands for this:

```
int int(StringData); // converts string to an int
float float(Stringdata); // converts string to a float
String str(numericalData); // converts a number to a String
```

We can use these to fix our above examples:

```
String s = "5";
int i = int(s); // convert the String to an integer
```

Likewise:

```
int i = 5;
String s = str(i); // convert an integer to a String
```

This fixes our problem, and these tools will be an important part of your toolkit.

Unfortunately, these commands aren’t that powerful. If the command gets confused, it just gives you a bad result. You need to watch for that. For example:

```
String s = "1,234";
int i = int(s);
float f = float(s);
println(i);
println(f);
```

In this case, the `i` gives you a 0, and the `f` gives you `NaN` – This is short for “not a number”. Yeah, THAT makes sense. Basically, if you get a `NaN`, it just means it’s broken.

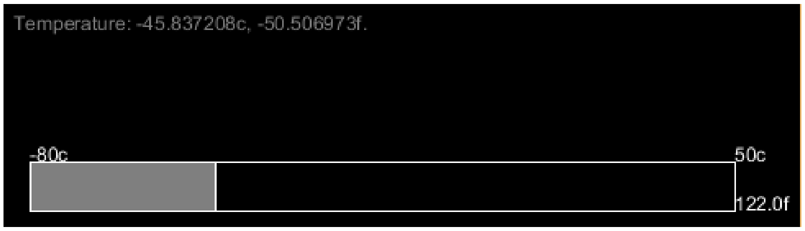
You will find that, in particular, converting from a number to a string for output is very common. This is so common, in fact, that Processing has a shortcut for it. If at any point, you combine a string and a number with a plus sign – that is, you try to concatenate a string with a number – Processing does the number-to-string conversion for you. For example:

```
String s = "My age: ";
s = s + 99; // actually s = s + str(99)
```

This is another nice thing that will save you a bunch of time.

8.5 Example: Celsius and Fahrenheit Scale

Let’s make an interactive Celsius and Fahrenheit scale, where you can move the mouse to select a temperature and see both the c and the f reading.



```
Temperature: -45.837208c, -50.506973f.
```

Let’s work through our globals to start setting up this project. Let’s start with the basic dimensions:

```
final int S_TOP = 100; // scale top
final int S_LEFT = 30;
final int S_WIDTH = 400;
final int S_HEIGHT = 30;
```

We also need to setup the range of the scale (hot to cold):

```
final int HOT = 50; // celcius
final int COLD = -80;
final int TEMP_RANGE = HOT-COLD;
```

Now that we have our basic setup thought out, we can start drawing our scale. Let’s use top-down programming to setup the draw block, defining the jobs that need to be done.

```
void draw()
{
  background(0);
  drawScaleBackground();
  drawScaleLabels();
  fillScaleUsingMouse();
  drawScaleReading();
}
```

Drawing the scale background just requires a rectangle with the correct colors, using the globals that we already setup. I use a white outline and black fill. This is simple, so I don't put the code here.

Next, we need the labels at the left and the right of the scales. This is slightly trickier, as there are three problems to solve. First, where on the screen are they drawn? Second, we need the Celsius and Fahrenheit for both, and this should be calculated and not hard-coded, as the `HOT` and `COLD` variables may change. Third, we need to construct the required string output.

The locations are not that hard. The leftmost top label actually goes at the scale left and top. Remember that text's y coordinates, when drawn, refer to the bottom of the string. So, if we want a string on the top of our scale rectangle, we use the same coordinates.

```
int x = S_LEFT;
int y = S_TOP;
```

Then, to calculate the actual Fahrenheit we use the standard formula. This was an exercise in the previous chapter. Make sure to use floating points, and be careful of integer division!

```
float f = 9.0/5.0*COLD+32;
```

To construct the labels, we use our conversion functions to convert the numbers to text, and then, use concatenation to attach the appropriate "c" or "f" suffix.

```
String celsius = str(COLD)+"c";
String fahrenheit = str(f)+"f";
```

Note: do you see a shortcut here? Concatenating a number with a string automatically does the conversion for you, so the `str` function calls are actually not necessary in this case.

Finally, we need to draw these. We have the x and y for the top label, and for the bottom, we just add the height of the scale. We also already have the strings:

```
text(celsius, x, y);
text(fahrenheit, x, y+S_HEIGHT);
```

Now that you have left labels, calculating the right ones is trivial. Try it on your own – you need new numbers, new strings, and a new location. Notice that you can reuse your existing variables for this.

Now, we have a nice scale with Celsius and Fahrenheit on it!!! Yay!

Filling the scale based on the mouse position is a little bit tricky, and requires the following steps to be done in our `fillScaleUsingMouse` function:

- Calculate how far along the scale the mouse is. Take the mouse position and subtract the left end of the scale
- Make sure we're not off either end of the scale!!
- Draw the filling using that width

```
int xOffset = mouseX - S_LEFT; // how far along scale
xOffset = max(0, xOffset); // if < 0, make 0
xOffset = min(S_WIDTH, xOffset); // if > width, make width
rect(S_LEFT, S_TOP, xOffset, S_HEIGHT);
```

Choose a color for the filling, too, before drawing that fill. I used grey. At this point, your program should draw a filling in the scale that animates as you move the mouse. It should be aligned with the mouse cursor's X position.

Finally, the last part is to draw the label at the top of the screen, which gives the current reading at the mouse position in both Celsius and Fahrenheit. This has several steps – we need to get how far the mouse is along the scale in pixels, then convert that to how many degrees that represents. Then we need to convert that to Fahrenheit, and construct our message.

We already calculated our mouse position along the scale in the `xOffset` variable in a different function. You can either re-calculate that here, or, make the previous variable global so that you can re-use the value. In that case, pay attention to the order that the functions are called in, to ensure that the variable is properly calculated first.

We convert the mouse offset along the scale (how far it is from the scale left) to a percentage of the scale first, and then mapping that to the temperature range.

First, we are hit with a problem. If we try to generate our percentage:

```
float tempPerc = xOffset/S_WIDTH;
```

Can you see the problem? Both `xOffset` and `S_WIDTH` are integers, giving integer division. We can fix this by making one of the variables, the `xOffset`, a `float`.

Now that we have the temperature as a percentage, we map it to our temperature range. First we find out where in the range we are, then we add in the minimum temperature.

```
float tempC = tempPerc*TEMP_RANGE+COLD;
```

Now we can also calculate the Fahrenheit:

```
float tempF = 9.0/5.0*tempC+32;
```

And construct our output message:

```
String message = "Temperature: "+tempC+"c, "+tempF+"f.";
text(message,20,20);
```

You should make sure to set the color, too.

All done! Here is my final code:

```
final int S_TOP = 100;
final int S_LEFT = 30;
final int S_WIDTH = 430;
final int S_HEIGHT = 30;
final int HOT = 50; // celcius
final int COLD = -80;
final int TEMP_RANGE = HOT-COLD;
final int xOffset = 0;

void setup()
{
  size(500, 500);
}

void drawScaleBackground()
{
  stroke(255);
  fill(0);
  rect(S_LEFT, S_TOP, S_WIDTH, S_HEIGHT);
}

void drawScaleLabels()
{
  fill(255);

  int x = S_LEFT;
  int y = S_TOP;

  float f = 9.0/5.0*COLD+32;

  String celsius = str(COLD)+"c";
  String fahrenheit = str(f)+"f";
  text(celsius, x, y);
  text(fahrenheit, x, y+S_HEIGHT);

  // draw right marks
  x = S_LEFT + S_WIDTH;
  f = 9.0/5.0*HOT+32;
  celsius = HOT+"c";
  fahrenheit = f+"f";
  text(celsius, x, y);
  text(fahrenheit, x, y+S_HEIGHT);
}

void fillScaleUsingMouse()
{
  // fill in thermometer
  int xOffset = mouseX - S_LEFT; // how far along scale
  xOffset = max(0, xOffset); // if <0, make 0
  xOffset = min(S_WIDTH, xOffset); // if >width, make width
  fill(127);
  rect(S_LEFT, S_TOP, xOffset, S_HEIGHT);
}

void drawScaleReading()
{
  // output the reading from the mouse
  float xOffset = mouseX - S_LEFT; // how far along scale
  float tempPerc = xOffset/(float)(S_WIDTH);
  float tempC = tempPerc*TEMP_RANGE+COLD;
  float tempF = 9.0/5.0*tempC+32;
  String message = "Temperature: "+tempC+"c, "+tempF+"f.";
  stroke(255);
  text(message, 20, 20);
}

void draw()
{
  background(0);
  drawScaleBackground();
  drawScaleLabels();
  fillScaleUsingMouse();
  drawScaleReading();
}
```

8.6 Characters

This may seem strange at first, but in Processing, in addition to the String type, we have a type for single characters. The reason for this is that while String is an object (complex!), the character is a primitive type. They are otherwise very like the other primitive types. In fact, Strings are actually internally made up of a collection of this character primitive type.

In the short term, we won't use characters much. Later, when we learn more advanced programming techniques, we will do more work with taking a string apart into individual characters.

To create a character variable, you use the keyword `char`, which can be pronounced like "car" (short for character), or char (like charbroiled).

```
char c;
```

You create a character literal by using the single quotes. Be careful, using double quotes gives you a string, which is different.

```
char c = 'j';
```

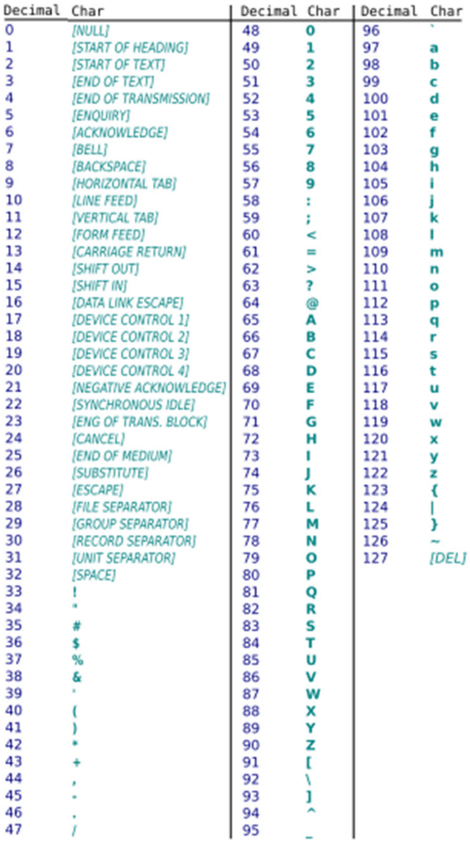
Characters can be numbers, letters, upper case, lower case, symbols, etc. Basically, anything that can go in a string, is a character. But be careful! You cannot place two characters in one variable:

```
char c = 'ja'; // doesn’t work
```

8.7 What is a Character anyway?

Everything in a computer is stored as a number. Characters are no exception. When early computer people started to store characters, they had to think up a way to convert a random character like the letter ‘q’ into a number. The solution was to develop a standardized lookup table, where each character would be assigned a number. For example, let’s say that ‘q’ is 113. Every time the computer encounters the character 113, it draws a q from its font. This kind of system requires standardization – all computers that talk to each other need to agree on this.

You can look it up, but no one knows what ASCII stands for. It’s an acronym, though..



Decimal	Char	Decimal	Char	Decimal	Char
0	[NULL]	48	0	96	`
1	[START OF HEADING]	49	1	97	a
2	[START OF TEXT]	50	2	98	b
3	[END OF TEXT]	51	3	99	c
4	[END OF TRANSMISSION]	52	4	100	d
5	[ENQUIRY]	53	5	101	e
6	[ACKNOWLEDGE]	54	6	102	f
7	[BELL]	55	7	103	g
8	[BACKSPACE]	56	8	104	h
9	[HORIZONTAL TAB]	57	9	105	i
10	[LINE FEED]	58	:	106	j
11	[VERTICAL TAB]	59	;	107	k
12	[FORM FEED]	60	<	108	l
13	[CARRIAGE RETURN]	61	=	109	m
14	[SHIFT OUT]	62	>	110	n
15	[SHIFT IN]	63	?	111	o
16	[DATA LINK ESCAPE]	64	@	112	p
17	[DEVICE CONTROL 1]	65	A	113	q
18	[DEVICE CONTROL 2]	66	B	114	r
19	[DEVICE CONTROL 3]	67	C	115	s
20	[DEVICE CONTROL 4]	68	D	116	t
21	[NEGATIVE ACKNOWLEDGE]	69	E	117	u
22	[SYNCHRONOUS IDLE]	70	F	118	v
23	[ENG OF TRANS. BLOCK]	71	G	119	w
24	[CANCEL]	72	H	120	x
25	[END OF MEDIUM]	73	I	121	y
26	[SUBSTITUTE]	74	J	122	z
27	[ESCAPE]	75	K	123	{
28	[FILE SEPARATOR]	76	L	124	|
29	[GROUP SEPARATOR]	77	M	125	}
30	[RECORD SEPARATOR]	78	N	126	~
31	[UNIT SEPARATOR]	79	O	127	[DEL]
32	[SPACE]	80	P		
33	!	81	Q		
34	"	82	R		
35	#	83	S		
36	$	84	T		
37	%	85	U		
38	&	86	V		
39	'	87	W		
40	(	88	X		
41	)	89	Y		
42	*	90	Z		
43	+	91	[		
44	,	92	\		
45	-	93	]		
46	.	94	^		
47	/	95	_		

Image cc, derived from commons.wikimedia.org/wiki/File:ASCII-Table-wide.svg

One early standard for this was called ASCII (pronounced ass-key). In ASCII, all the basic characters and some “control” characters (to send commands to, e.g., old printers), were put onto this table. Everyone agreed, so now that every time the computer encounters the number 84 as a character, it knows that it’s a capital T.

Advanced you can test this out. If you force a character to be read as a number, then Processing will tell you the ASCII number. How can we do this? We can just store the character into an integer. There is some things happening under the hood here that we’ll learn soon. People don’t really do this, though, so it’s just a bit of a toy example.

```
char c = 'X';
int number = c;
println("The ASCII number for "+c+" is: "+number);
```

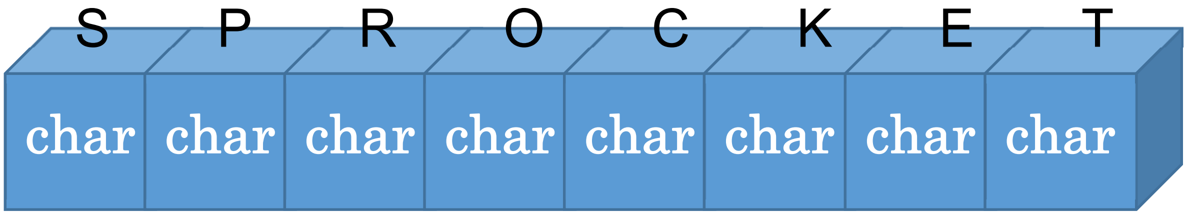
Looking at the above ASCII table, can you see any limitations? This is extremely limited! No accented letters! Does not handle complex writing systems! We need a new standard that can handle Chinese, Arabic, Hebrew, and Korean!

こんにちは！ 中国語 안녕 하세요 שָׁלוֹם

A new standard was developed for all languages – it's called Unicode. One code to rule them all, one code to bind them. We don't cover Unicode in this course, but it's good to know a rough idea of what it is.

8.8 Characters and Strings

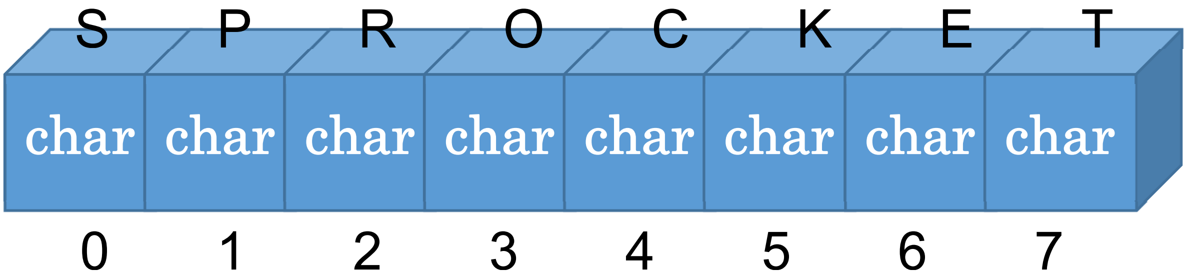
So I told you that Strings are made of characters, so now let's learn a little more about that. You should think of strings as a series of boxes, where each box is a character. For example, the string "SPROCKET" is stored internally as follows:



You can see that each character in the string gets its own box. Even spaces, symbols, etc. Now, internally Processing numbers them in a very specific way that is simple at first but will end up causing you all kinds of grief:

String Character numbering: the characters are numbered in order, starting at 0.

As follows:



How many characters are in the string "SPROCKET"? 8. What is the index of the last character? 7!!!!!! This is called the dreaded off-by-one error which will plague you for the rest of your computer science career.

Off by one error: Since computers start counting at 0, and we generally start counting at 1, your intuition is often off by 1.

For example, with strings, the index of the last bin is always the length of the string -1, as above.

8.9 **String methods**

Strings, since they are objects, can be used differently than other variables. There are a new kind of command that we can do on Strings. Up until now, we call commands functions. When a command is attached to an object, we call them methods. Here is some syntax (not exactly how you use it, that's next):

```
int stringVariable.length(); // number of characters
char stringVariable.charAt(int index);
```

These are commands that can tell you how long a string is in number of characters (including spaces, punctuation, symbols, etc.), and, that can get you a character in a specific bin.

These commands can be used as follows:

```
String s = "Hello world!";
println(s.length());
println(s.charAt(0)); // first character
```

The output here is 12, and H. 12 is the number of characters in the string, and H is the character at bin 0, which is the first bin. How would you get the last character using these commands?

```
s.charAt(s.length()); // ERROR! Off by one
s.charAt(s.length()-1);
```

For now, you will mostly use strings to put text on your screen relating to your program. Soon, however, we will learn more advanced techniques that will let us do things more with the character and length methods, for example, calculating someone's initials.

Check your Understanding

8.10 Check Your Understanding: Exercises

Exercise 1. Computers often use template text in a situation, like a website, and then populate it depending on the user. Write a program to use the following template text:
"Hello <name>, from <city>, <country>. Thank you for shopping at <store>."

a. Create variables for each unknown, and assign them reasonable values.
b. Using string concatenation, construct a single new string with the entire message in it
c. Put the string to console
d. Put the string onto a canvas graphically

Exercise 2. Make a program that takes two digits, as Strings, such as "1" and "5" (call them number 1 and number 2), and create two numbers by concatenating them in both orders. In this case, you get "15" and "51". Convert them to numbers, and divide the first (number 1 then 2) by the second (2 then 1), in this case, 15/51. In what instance do you get 1 as a result? In what instance may this crash?

Exercise 3. Your friend is having trouble understanding the coordinate space in Processing, so you came up with a plan to help them. Make a program that displays the mouse's current position on the screen, so that as the mouse moves around they can see how the position changes.

a. Construct a string that stores `"<mouseX>, <mouseY>"`
b. Set the text size to 20
c. Display the text at x=0, and half way down the screen

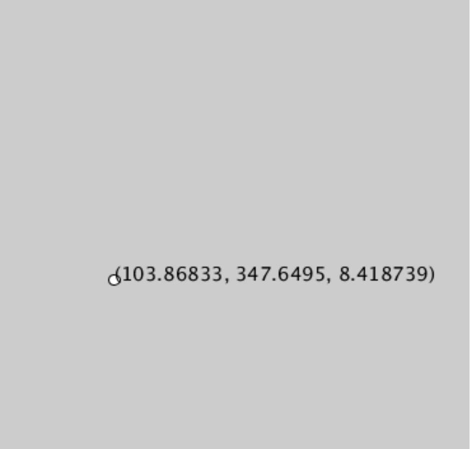
Exercise 4. The same friend has trouble understanding how the font sizes change. Update Exercise 3, and make the font size equal to the mouse's y coordinate, divided by 5.

Exercise 5. Make a program that prints out a string to the console in a special formatting. Given a string variable called data, construct the following: `"-<firstCharater>-<string>-<lastCharacter>-(<length>)″` and output it to console. For example, if data was set to "Hello!", then the output would be "-H-Hello!-!-(6)".

Exercise 6. Update Example 8.5, the temperature slider, to remember the maximum temperature that the mouse has ever reached. At that spot, draw a line through the scale, and put the text “Max: <maximum value>”.

Exercise 7. Make a program that plots a circle at a random spot and size on the screen, and displays the coordinates and size of the circle. How would you get whole numbers only, and not real numbers? We haven’t learned how yet ☺.

a. Move the text slightly away from the circle, up and to the right.



```
(103.86833, 347.6495, 8.418739)
```

Exercise 8. Make a program that turns a number into a special circle on the screen. Given a number, first get how many digits that number has (hint: convert to a string). Then, the x coordinate of the circle is the length to the power of 4, modulo the screen width. The y coordinate is the length times the number itself, times the last digit in the number, modulo the height. Make the ellipse 50x50. (The output here is pretty boring – but as long as it works reasonably well, you are good to go. Just for practicing all the new stuff you learned).

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 9. DATA TYPES AND MEMORY

Summary

Here you will learn the basics about bits and bytes, and how computers store data. You will also see how to convert between similar numeric types.

In this section, you will…

- Learn that some data types take up more memory than others.
- See nearly the full list of primitive data types.
- Learn about casting, converting between data types.

Learning Objectives

After finishing this unit, you will be able to …

- Make integer or floating point variables that use more or less memory.
- Convert between data types using explicit and implicit casts.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

9.1 Introduction

Now that you have some programming practice let's take a step back and learn a little about bits, bytes, computer memory, and what this means for your data types. We will learn about how data is stored (only briefly!), what different data types are available, and how you go between the data types.

9.2 Bits and bytes and nibbles

Advanced: A computer stores everything as switches that can be either on or off. These switches are called bits, and usually are written as a 0 for off, or a 1 for on. One bit has 2 possible settings. A combination of two bits has 4 possible settings `(00,01,10,11)`. In fact, *n* bits has 2^n possible settings. If you stick with computers, you'll get very comfortable with powers of 2, for this reason.

How do you count with bits? Logically, it follows the same as when you count in regular base-10 numbers, but it takes a bit to wrap your head around it. The first two numbers are easy

```
0
1
```

But now, since we ran out of digits (binary only has `0,1`), we need to add a new column on the left, and add a 1. This is like when we go `0,1,2…9`; if we want to count higher, we add a 1 on a new column to the left, and reset the rest of the number to zero, to get a 10.

```
10 // 2
```

now we just add a 1 again to get 3

```
11 // 3
```

Like above, we ran out of digits so we reset to zeros and add one on the left. Just as 99 becomes 100, then 11 becomes 100 in binary

```
100 // 4
```

Try to keep going with this. How would you write 31?

A group of 8 bits (switches) is called a byte of memory

```
00110101 ← one byte of data
```

A group of 4 bits (half a byte) is called a nibble (seriously!!)

```
0110 ← a nibble of data
```

From here we scale up.

2^{10} bytes, or 1024 bytes, (8192 bits) is called a kilobyte

2^{20} bytes, or 2^{10} (1024) kilobytes, make a megabyte

2^{30} bytes, or 2^{10} (1024) megabytes, make a gigabyte

2^{40} bytes, or 2^{10} (1024) gigabytes, make a terabyte

A terabyte has 1,099,511,627,776 bytes. It has 8,796,093,022,208 switches (bits).

If we line up those many switches using standard 7 cm light switches, we get a line of switches 615 million kilometers long! That is four times the distance from the earth to the sun. Holy cow, that's a lot of switches.

As an aside, some new standard units are moving to even powers of ten, where 1 terabyte = 1,000,000,000,000 bytes. This is not easily represented inside a computer because it's not a power of two, but people like it better.

9.3 **Using More or Less Memory**

Variables use computer memory, and each variable type has a clearly defined amount of memory that it can use. A variable that uses more memory can store a wider range of numbers, and/or more precision, and a variable with less memory can store less. Although our examples are very small and you may not think you need to worry about computer memory, professional programmers are generally thinking about how they are using memory and keep an eye to being economical.

Let's look at the `int` data type. As we learned previously, the largest number that an `int` can store in Processing is 2147483647 and the smallest number is -2147483648. This seemingly-random number is actually determined by how the number is stored; ask your professor if you are interested:

```
int number = 2147483647;
```

Now, what happens if we increase the number by 1? To try and force it to store that bigger number?

```
number++
println(number);
```

If you try this, you will see that you get a very large negative number. In fact, what we are seeing here is hitting the memory limits of the integer data type. If we go over the limit, the values roll over. The same happens in the other direction – if we keep going minus, we will see eventually a positive number.

This can be very confusing! If you end up seeing unexpected negative numbers when you are doing calculations, it may be because you had an overflow.

If you come across overflow, you generally need to try and be cleverer with your numbers to avoid such large numbers. Alternatively, you can use a data type with more memory.

9.4 The Primitive Data Types

There are only a few basic, core data types, in most programming languages. We have already seen three basic types: `float`, `char`, and `int`. Remember, `String` is an object and acts differently. This actually covers most of the general territory of primitive types. We will soon learn boolean, a type for true and false only. However, in addition to this, we mainly have variants of the integer and floating point types with more or less memory. For example, in the integer class we have the following variables

type	size	minimum	maximum
`byte`	1 byte	-128	127
`short`	2 byte	-32,768	32,767
`int`	4 byte	-2^{31}	$2^{31}-1$
`long`	8 byte	-2^{63}	$2^{63}-1$

You can use these types just like the other ones we have learned. Also, these are integers, so they follow the same rules regarding integer division, etc.

For the most part, you will only use the larger type (long) if you want a larger range of numbers to store. Except for special circumstances, there is little benefit to using the smaller types.

Advanced: You may think you can save memory by using, for example, the short type. However, you won't likely save memory. Computers work fastest on number sizes that match their processor. For example, in recent years you hear about 32-bit and 64-bit processors. If you have a 2 byte short variable, and a 32 bit computer, your computer is likely to store those 2 bytes alongside 2 empty bytes to make a 32 bit number, to save time and processing to work with it. So, even though a short has 2 bytes, your computer still will probably use 4 to store it. In exceptional circumstances (not in Processing or Java), you can tell your computer how to pack these numbers, for example, when you really need that memory saved, or you need to ensure how many bytes are used.

For floating point numbers, we only have two types. We have the `float` (4 bytes) and the `double` (8 bytes) types. Unlike integers, more memory in a floating point does not manifest itself only a larger range. Instead, it is also *more precision*. For example, if you perform 2/3 you get the following results:

`float` - 0.6666667

`double` - 0.6666666666666666

As you can see, the double variable gives a closer approximation. In general, we will stick to the `int` and `float` types, but you should be aware of these variants.

Advanced: `float` or `double`? Which is better? Clearly, double has better precision. If you have a 64 bit machine, your 32 bit (4 byte) `float` will likely take 64 bits of memory anyway, so why bother with `float`? If you talk to people who do scientific computing, they will argue strongly to always use `double`. In fact, in regular Java, `double` is the default floating point type. However, if you talk to a graphics person, they may tell you to always use float. Also, processing defaults to `float`. Why is this?

The complication here is graphics cards – your computer has a specialized processor to do graphics. Even if you spend a boatload of cash on a high-end gaming video card, it is very likely optimized for `float` and not `double`. For graphics, speed is more valuable than a bit of extra precision.

9.5 Going Between Data Types: Casting

If you take a metal statue – like a miniature lion – and you want to turn it into a statue of a dog, one way to do it is to first melt the lion down (it's metal, after all), and pour it into a mold, a cast, of a dog. Once it cools, now you have a metal dog. This is called casting, when you take a metal and force it into the shape of the cast.

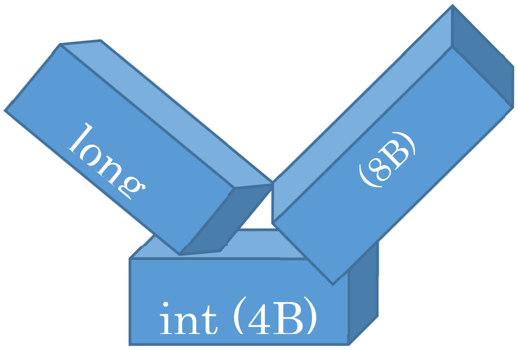
There is a similar idea in programming. Casting is taking one kind of data and forcing it into the shape of another type. Although we haven't dealt much with any integer type except for `int`, let's try a few things.

```
int i = 1234;
byte b = i; // store 1234 in b
```

Processing complains and won't compile: `cannot convert int to byte`. We may have guessed this, since we know that byte can only go as large as 128 and cannot store 1234 (check the table on the previous page). Let's try another one:

```
long l = 1234;
int i = l;
```

Even though we know that the integer can handle the number `1234`, Processing still

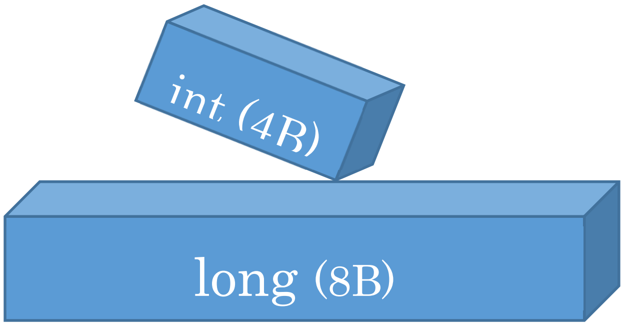


complains: `cannot convert long to int`. In fact Processing does not even look inside to see if the number fits. It just knows that the datatypes are of different sizes, so it complains. The issue here is that you can lose data mistakenly. What if the long actually had a number that was too large for the integer? What should happen then? Should it roll over like in our earlier examples? It is quite confusing, so to play it safe, Processing just stops and doesn't let you do it. This is called a ***narrowing conversion***, since you are moving from a more-capable type to a less-capable type. You can lose data if you are not careful: the diagram illustrates this, showing that you need to break a long up (8 bytes) to make it fit into the 4 byte box.

What about the other direction? What if we want to store an `int` type into a `long`? Or a `byte` into an `int`? Try the following

```
byte b = 12;
int i = b;
long l = i;
```

Here, we take a `byte`, then try and store its value into a 4-byte `int`, and then try to store that into an 8-byte `long`. If you try these in Processing, they all work just fine. This is because we are asking Processing to move from a less capable type to a more capable one. This is called a ***widening conversion*** – as in the inset below, Processing has no problem with this since there is no risk of data loss. The 8 byte `long` can easily accommodate the 4 byte `int`. When data is converted in this way – from a less-capable to a more-capable type – it is called an ***implicit cast***. The conversion happens implicitly, without you asking for it.



Sometimes, you want to force a narrowing conversion even though you know that data may be lost. For example, you may be working in a `long` datatype to work with large numbers but you know that your result is small enough to fit into an `int` (perhaps you divided it by a large number). In this case, you need to use an ***explicit cast*** to make this happen – you need to tell Processing that you know what you are doing and you want it to cast the data to the new type, even though data may be lost.

Let’s do an example:

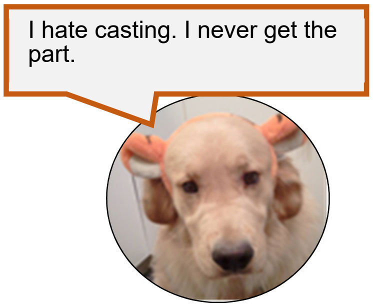
```
long large = 200;
int small = large;
```

In this case, Processing will not run this code because of the narrowing conversion. However, we know that it is safe, so we want to force a conversion using an explicit cast. The syntax of an explicit cast is as follows:

```
(newType)data
```

For example, we can update the above example:

This tells processing that we know what we are doing, and to go ahead and do the conversion.



```
long large = 200;
int small = (int)large; // explicit cast
```

The same logic above holds for going between floating point types. Since `double` uses more memory than `float`, it is larger. Going from `float` to a `double` is a widening conversion, and can be done with an implicit cast. Going from `double` to `float` is a narrowing conversion and requires an explicit cast:

```
float f = 1.23;
double d = f; // implicit cast OK!
d = 5.123
f = d; // ERROR! Cannot convert, narrowing conversion
f = (float)d; // OK - explicit cast.
```

What about going between the integer and floating point types? In this case, we need to think about it a little differently. Instead of thinking about more or less memory, think about more or less capable. A floating point number can store more detail than an integer, which is limited to whole numbers only. Therefore:

- ***Going from integer -> floating point is a widening conversion, and does not require an explicit cast.***
- ***Going from floating point -> integer is a narrowing conversion, and requires an explicit cast.***

When you go from a floating point to an integer, you need an explicit cast. Java just cuts off the decimal portion and stores the whole number. For example:

```
float f = 1.234;
int i = (int)f;
println(i)
```

What do you think the output will be? Try it out.

One more thing to mention is the order of operations with casts. What about the following?

```
int i = (int)0.5*3.0;
```

What do you expect will happen? If you try to run this, Processing complains that it cannot do the conversion. This is because ***casts happen first in order of operations***. In this example, the 0.5 gets converted to an `int` first, which becomes a zero.

```
int i = 0*3.0;
```

then there is a floating point multiplication, resulting in 0.0, which cannot be stored in an `int`, since it is floating point.

In this class, you will primarily use casting to go between integer and floating point numbers, but you will need to understand the basics of how it works in general, as it is testable. You will use casting a great deal more in future courses in two ways: one, you will do more bits-and-bytes work, and two, casting ends up showing up in object oriented programming, as well.

There are two final notes about casting. First, you can use casting to go between integers and characters, as long as you understand that a character is just a number underneath. Two, when you convert between String and numerical types, we do not call it casting, because a lot of work goes into it; data just isn't forced into a new type (with possible data loss). Someone actually wrote a program to analyze your data and convert it to a string or number.

Check your Understanding

9.6 Check Your Understanding: Exercises

Exercise 1. Make a program to help a shipping company figure out how much oil can fit inside a square tanker. The tanker has dimensions of width: 1234mm, height: 5678, and length: 9012. This company uses barrels that store 1 million cubed millimeters.

a. Calculate the volume of the tanker using integers, by multiplying the width by the height by the length.
b. Calculate how many barrels can be stored by dividing the total volume by a million.
c. `println` the result. What is wrong with it? How can you fix it? Your final answer should be 63143 barrels.

Exercise 2. Your friend wrote the following code to generate a random dice roll (on a standard 6-sided die), but complains that a) they get decimal numbers, and, the dice roll starts from 0 instead of 1.

```
prinln(random(6));
```

a. You know that the decimals happen because `random` gives you a floating point result. How can you fix this using casting?
b. Fix the off-by-one error

Exercise 3. The following code contains two casts: an explicit and an implicit one. Can you identify both?

```
float f = (int)PI;
```

Exercise 4. Up until now, you can avoid the problems of integer division by making one of the operands a float; remember, if both operands are integers, the computer does integer division. Without changing the data types, and only introducing casts, how can you fix the following code bug which unfortunately uses an integer division?

```
int units = 10;
int dollars = 5;
float cost = dollars / units;
println("the cost is: "+cost+" dollars each");
```

Exercise 5. Use casting and a little bit of math to print out a floating point number to two decimal places only. For example, `println(PI)` should give you 3.14.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

UNIT 10. TRUE AND FALSE – BOOLEANS

Summary

This section covers boolean logic, how computers work using true and false. You will learn how to store and calculate Booleans, as well as how to make your program do different things based on Booleans.

In this section, you will…

- Learn about booleans, how computers work on true and false values instead of numbers.
- Learn how to make your program do different things based on the conditions
- Learn basic boolean operations such as *not*, *and*, and *or*.
- See how variable scope is impacted by conditional programming

Learning Objectives

After finishing this unit, you will be able to …

- Create boolean variables and store `true` and `false` in them
- Use boolean variables in `if` statements to have conditional programs: actions that depend on a boolean test.
- Use `else` blocks to have code run when a condition is not true
- Perform basic boolean operations, the `not`, `and`, and `or` operations.
- Use nested `if` statements, when one conditional is inside another.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

10.1 **Introduction**

It’s time to learn a new data type: let’s learn about Booleans!

While integers and floats can have billions of different values stored in them, ***boolean variables can only have two different values: true and false***. You make them with the `boolean` keyword.

```
boolean jimIsRich = false;
boolean teoIsCute = true;
```

And, as shown in this example, you can store `true` or `false` just by typing those words. Be careful, they need to be all lower case.

So far, it may not be clear why this is useful. But now that we have booleans, we can do conditional statements!!! Program code that only runs if some condition is true.

10.2 **Basic Conditionals**

So far, our programs run in a pretty straight path. We put a series of commands into Processing, and it runs them straight through in the same order every time, and runs every single command. What if we only want commands to run some of the time? Remember our early example of drawing lines behind the mouse to make a paint program? What if we only want to draw when the mouse button is pressed? What if we want to erase the screen after so many lines? What if we want to change the color based on where it is drawn?

All of these require a new way of thinking about programming. They require an “if X then do Y” type of logic that is not possible with the tools we learned so far. We need a way to tell processing which lines of code are run and which are not, and under which conditions. Here is the syntax in processing:

```
if (booleanValue)
{
  // then do this!!
}
// more program
```

Be careful: the boolean value is in round brackets, and it is followed by a block of code in curly brackets. With this syntax, if the boolean value is `false`, the entire code block is skipped. If the boolean value is `true`, then the code block is executed. So far, we haven’t learned how to do much with Booleans, except store `true` or `false` in a variable, so this looks boring. I’ll show you some great built-in boolean values that processing offers, to make things a little more interesting.

```
boolean mousePressed; // true if mouse button is down
boolean keyPressed; // true if a key is being pressed
```

These are built in variables that processing offers. Before starting the `draw` block each time, processing will check the mouse and keyboard, and set those variables to `true` or `false` accordingly.

Let's try it out! Try the following program:

```
void draw()
{
  println(mousePressed);
}
```

This will continually print the status of the mouse button. It should start by printing a series of `false` to the console. If you click on the canvas (it has to be on the canvas), and hold the button down, you'll see the `true` come up.

Note that this tests if the mouse is currently pressed at that `draw` block. It will be `true` as long as you are holding it down. Even if you try to click very quickly, it will probably be `true` for several draw cycles.

Next, let's take our previous example of a drawing program. I'll copy it here:

```
void setup()
{
   size(500,500);
   background(0);
}
void draw()
{
   stroke(255);
   line(pmouseX,pmouseY,mouseX,mouseY);
}
```

Type it up and make sure it works. Let's update this so that it only draws when the mouse button is pressed. Remember, we use the `if` syntax above:

```
if (mousePressed)
{
   line(pmouseX,pmouseY,mouseX,mouseY);
}
```

Notice that, like with other blocks, I indent this as well. Indentation cumulates to improve readability (check the full program on a later page).

In this case, each time draw runs, your code will check if the mouse is pressed. If that variable stores false, then the block is skipped. When it has true, then that block is executed. Let’s add another condition. Let’s clear the screen if a key is pressed on the keyboard!

```
if (keyPressed)
{
 background(0);
}
```

Now we have a much more functional program!! These lines of code *inside* the if blocks only run when the booleans are true. When they are false they are skipped. Also keep in mind that, because it is a code block, you can put many commands inside there just like any other block. Here is my whole program

```
void setup()
{
  size(500,500);
  background(0);
}
void draw()
{
  stroke(255);

  // only draw when pressing the mouse button
  if (mousePressed)
  {
    line(pmouseX,pmouseY,mouseX,mouseY);
  }

  // erase the screen if any key is pressed
  if (keyPressed)
  {
    background(0);
  }
}
```

Warning: there is a huge, very common bug with `if` statements. Up until now in the course, you put a semi colon after most lines. However, just like with functions, you don't put a semi-colon after an `if` statement. Unfortunately, if you do put a semi colon, the program will still run, but incorrectly. Consider the following example:

```
if (mousePressed);
{
  line(pmouseX,pmouseY,mouseX,mouseY);
}
```

Do you see that extra semicolon? If you run this code, it will run, but work incorrectly. From the computer's perspective, that semi colon means end of command. So, the computer reads to itself: If the mouse is pressed, end of command (do nothing). OK. Next, do that whole block and draw the line. In this case, the line is drawn regardless of the value of mouse pressed. The block is not connected to the `if` statement and is always run. This is very confusing: ***if you have an*** `if` ***statement that seems to always run, check for an extra semi-colon***.

10.3 **Boolean Negation**

So you now know how to do something IF a boolean is true, but how can we do something if a boolean is not true? For example, in our drawing program, what if we want to detect if the mouse is *not* pressed, and put a black circle under the mouse at that time, to erase what is under it? How can we say - if the mouse is NOT pressed?

We need the ***negation operator. The negation operator takes the opposite of a boolean value***. If we have true, it gives us false. If we have false, it gives us true. All that we have to do is to put a `!` in front of it, the exclamation mark:

```
!boolean  // gives you the opposite of a Boolean
```

For example, instead of doing something when the mouse is pressed, we can do something when the mouse is not pressed

```
// if opposite of mousePressed is true
// --> if mousePressed is false
if (!mousePressed)
{
  stroke(0); // change the drawing color to black
  fill(0);
  ellipse(mouseX, mouseY, 50, 50);
}
```

Now you have an eraser! You can just scrub over anything that you drew to erase it.

You can put the negation operator (`!`) in front of any boolean variable or value, in any context, and it will give you the opposite.

10.4 **If-then-else**

This pattern that we just saw – do something if true, do something else if not true, is very common in programming. Processing provides special syntax just for this as per below. After the `if` block, use the keyword `else`, and provide a new block. The latter block is only run if the test is false:

```
if (boolean)
{
  // do if test is true
}
else
{
  // do if test is not true
}
```

Try on your own to use this new technique to update our program above.

10.5 **Nested if statements**

Once you are inside an `if` or an `else` block, you can freely add another `if` statement inside. This `if` statement is part of the block, so it only gets considered if the outside test is true. Currently, if the user presses a key the screen gets cleared to black. Let's make it so that if they press a key AND the mouse is down, then the screen gets cleared to white. First, we check if the key is pressed. If that is true, and we're inside that block, then we use a nested if statement to check if the mouse is pressed.

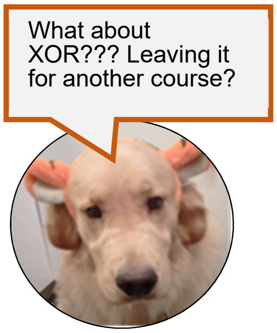
```
if (keyPressed)
{
  // this is ONLY checked if the above keyPressed is true.
  // Otherwise, the test is never even run.
  if (mousePressed)
  {
      background(255); // mouse and key pressed
  }
  else
  {
      background(0); // key pressed but not mouse
  } // end the inside else block
} // end the outside if statement
```

This can get complicated very quickly. The trick to getting this is to know what happens at each step, and realize that the computer steps through blindly and doesn't look at the whole program. In this case, if the first if statement is false, it jumps and skips everything. The inside stuff is *only run* if the first test I true. Again, if the `keyPressed` boolean is false, then the whole block gets skipped and no internal `if` statement is ever checked.

You can nest as many `if` statements inside `if` statements as you want. This can get very deep. Usually, however, if you nest deeply it becomes very confusing so we try hard to avoid that to keep things clear.

10.6 **Logical Operators**

In the above example, we kind of combined multiple booleans. We said if THIS is true, and, if THAT is true, then do something. Processing has logical operators that help with this kind of problem. For example, the above program could be done a little more cleanly, but so far, we can do this kind of AND logic by nesting `if` statements. However, there is a better way to do it. We can combine two booleans using an AND or an OR operator. This may be a little confusing, so let's look at the syntax first. The following two operators, the AND and the OR operators, take two booleans, and produce a new one depending on their values.



`booleanA && booleanB`	AND, true if both are true
`booleanA || booleanB`	OR, true of one is true

The AND operation uses two ampersand signs, commonly known as AND signs. The OR operator uses two pipe symbols, which you may be less familiar with. Take a moment to familiarize yourself with these on your keyboard.

We can use these to generate new booleans. For example, if we want a program to draw only if the mouse is pressed and a key is pressed, then we could do something like the following:

```
boolean shouldDraw = mousePressed && keyPressed;
if (shouldDraw)
{
  line(pmouseX,pmouseY,mouseX,mouseY);
}
```

In this case, if `mousePressed` is true, AND, `keyPressed` is true, then `shouldDraw` is true. Otherwise, `shouldDraw` is false. Then, if `shouldDraw` is true, we draw our line.

We don't actually need the extra variable here, and the logical operator can be used directly within the `if` statement:

```
if (mousePressed && keyPressed)
{
  line(pmouseX,pmouseY,mouseX,mouseY);
}
```

Although this boolean logic seems quite simple, it gets confusing fast, and it causes a lot of trouble for people. In particular, we do not think this way in daily life, and so we often get tripped up. The following charts may be useful; they are called truth tables, which give the full rundown of what different boolean values give you as a result with the operator.

AND (&&) truth table

A	B	Operation	Result
false	false	A && B	false
true	false	A && B	false
false	true	A && B	false
true	true	A && B	true

OR (||) truth table

A	B	Operation	Result
false	false	A || B	false
true	false	A || B	true
false	true	A || B	true
true	true	A || B	true

10.7 **If Statements, Blocks, and Scope**

We briefly discussed scope rules regarding variables, and the different blocks of our program. As a reminder, ***scope is the range within which a variable exists.*** Outside of that scope you cannot access or work with that variable. In fact, a variable gets destroyed and loses all its information once the program leaves the scope.

Also as a reminder, the scope of a variable is the block it was created in. So, variables created within one block can ONLY be accessed within that block. Each code block has its own local scope. This was true for functions, and is also true for `if` statements. For example, try the following code:

```
boolean hasUsedScopeMouthWash = true;
if (hasUsedScopeMouthWash)
{
      int freshness = 10;
}
print(freshness);
```

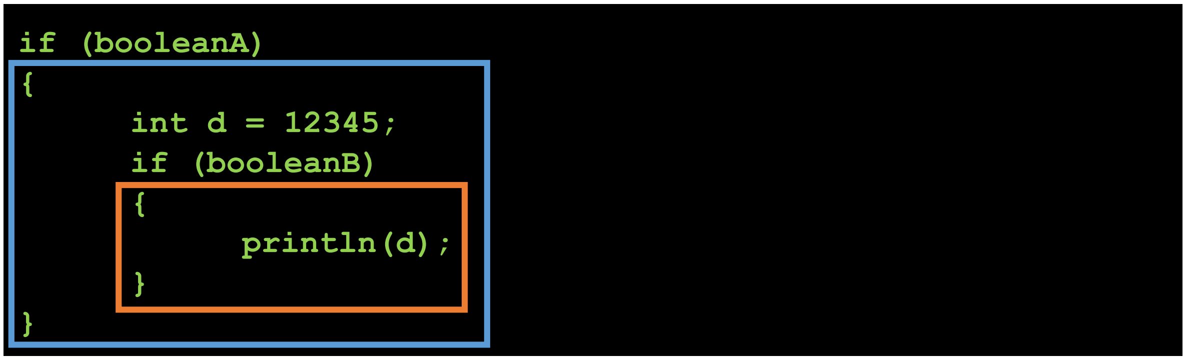
It doesn’t compile! Processing says: `Cannot find anything named “freshness”` on the print command. This is frustrating because you can see that you created the variable just above that line. The issue is that the freshness variable is created inside the `if` statement block. Once that block is over, the variable is said to be *out of scope*, and is destroyed. How do you fix this?

You can fix it by moving the declaration outside of the `if` block. If you declare the variable beforehand, then it is created in the broader scope and not limited to the `if` statement. Be careful: if you create the variable outside the block, you need to give it an initial value. ***Why? Try it out, and think about on your own. Mentally work through all the examples. As your instructor for clarification.***

What about nested blocks? Let’s look at the following:

```
if (booleanA)
{
      int d = 12345;
      if (booleanB)
      {
            println(d);
      }
}
```

In this case, does the print statement work? Yes – because the nested `if` statement is *inside* the scope of the outer one. Consider the example with highlighted blocks:



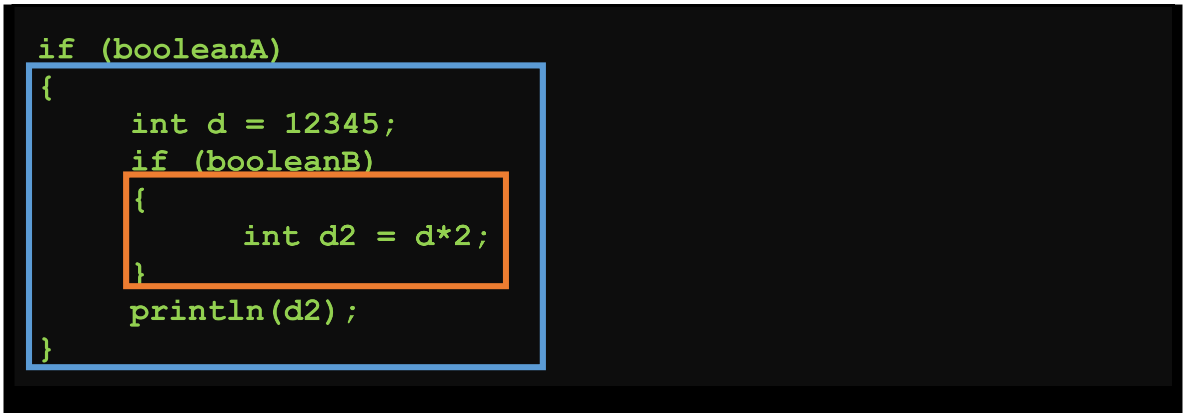
```
if (booleanA)
{
      int d = 12345;
      if (booleanB)
      {
            println(d);
      }
}
```

As you can see, the outer `if` statement scope is defined by the blue box. The inner scope, defined by the orange box, is completely within the blue box, so it is within the scope. Since the `int d` variable is created within the blue box, and the orange box is within the blue box, the print statement can see the `d` variable.

What about this similar example?

```
if (booleanA)
{
      int d = 12345;
      if (booleanB)
      {
            int d2 = d*2;
      }
      println(d2);
}
```

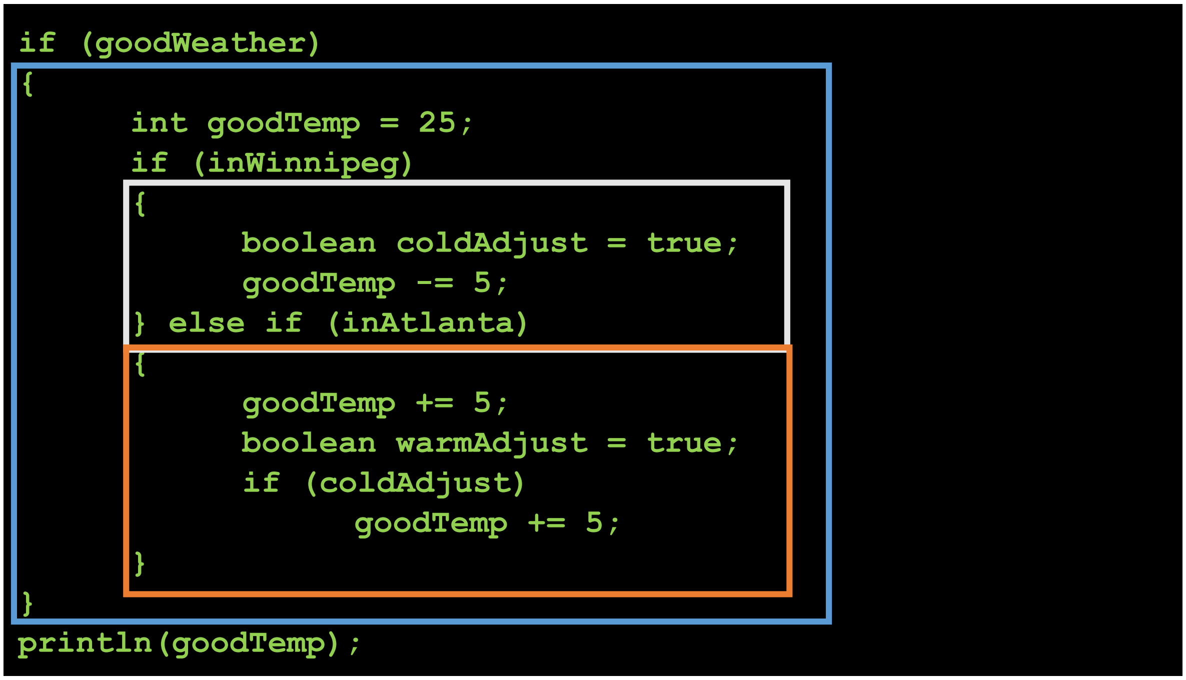
In this case, there is a scope error, on the `d2` variable. You can see this in the following copy with the scope highlighted:



```
if (booleanA)
{
      int d = 12345;
      if (booleanB)
      {
            int d2 = d*2;
      }
      println(d2);
}
```

Since `d2` is created within the orange scope, of the internal `if` statement, it gets destroyed as soon as the orange block ends. Thus by the time print tries to access `d2`, it is no longer available.

Scope can actually get quite confusing, quite quickly. Consider this hypothetical code for calculating what may be considered a good temperature based on what region you are in, with code blocks highlighted:



```
if (goodWeather)
{
      int goodTemp = 25;
      if (inWinnipeg)
      {
            boolean coldAdjust = true;
            goodTemp -= 5;
      } else if (inAtlanta)
      {
            goodTemp += 5;
            boolean warmAdjust = true;
            if (coldAdjust)
                  goodTemp += 5;
      }
}
println(goodTemp);
```

Can you spot the two errors? `goodTemp` is out of scope at the print statement, since it was created in the blue scope. The test on `coldAdjust` is out of scope on the `if` statement, since it was created in the white scope.

How to avoid scope issues: ultimately, you will develop a sense and clear understanding for scope. In the meantime, try to declare your variables at the top of the program, the top of the draw block, or the top of the function for now.

Check your Understanding

10.8 Check your Understanding: Exercises

Exercise 1. Create an active Processing program that draws a line from the origin to the mouse position, but only when the mouse is pressed.

Exercise 2. Create an active Processing program that draws random lines on the screen if a key is pressed on the keyboard, and clears the screen if a mouse button is pressed.

Exercise 3. Create an active Processing program that draws a white circle under the mouse if a key is pressed on the keyboard OR the mouse is pressed. Otherwise, draw a black circle under the mouse.

CuSn Exercise 4. Create a drawing program as introduced in this chapter, that draws a solid line under the mouse if a mouse button is pressed. In this case, if a key is being pressed on the keyboard, draw a dashed line under the mouse instead. You can make a dashed line by alternating which color is drawn each time, between white and black. Hint: You can use this using a boolean variable and an `if` statement.

Ag Exercise 5. Create an active Processing program that draws a circle in the middle of the canvas, that slowly grows each frame. If the mouse is pressed, instead of growing, the ball should shrink.

Ag Exercise 6. Make a program that enables a user to steer a square around the screen. This control scheme is quite hard to control.

a. If a mouse button is pressed, the square moves left. Otherwise, it moves right.
b. If a key is pressed on the keyboard, the square moves down. Otherwise, it moves up.

Ag Exercise 7. Make a program similar to the above exercise of moving a square around, except this time, you move a circle. The circle has a movement angle and speed that is stored globally. You will need to use your basic trigonometry.

a. If the mouse is pressed, the angle increases.
b. If they keyboard is pressed, the angle decreases.

Ag Exercise 8. Make a program that draws a square at a random location on the screen. The square is normally white. If the keyboard OR the mouse is pressed, but not both at once, the square should be black. If both are pressed, it should be white. This should use a single if statement and no nesting.

Au Exercise 9. A problem with your Processing programs is that your mouse and keyboard variables `mousePressed` and `keyPressed` tell you only that it is currently down. What if you want something to happen *only once* when the mouse is pressed? For example, place a circle under the mouse only once? If you make a program to draw a circle when the mouse is pressed, it will draw it each frame, and will animate as you move the mouse.

a. Make a program that first detects when the mouse is pressed and released. When the mouse is pressed, write “mouse down” *only once* to the console. Nothing else should be printed until the mouse is released. At that point, print “mouse up”. Hint: use a global to remember the last mouse state at watch for transition.

b. Now that you can detect a mouse press, draw one circle under the mouse each time it is clicked.
c. Update the program to similarly detect when a key is pressed.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 11. Advanced Conditionals

Summary

Now that you have basic conditionals under your belt, we can look at more advanced techniques. In this section, you will…

- Learn about how to compare data, e.g., if they are equal
- Learn exception cases including comparing strings and floating point numbers
- See new syntax for chaining multiple if conditions together
- See how code blocks are not always needed (but should be used)

Learning Objectives

After finishing this unit, you will be able to …

- Compare numerical values, and see if they equal, or if one is larger than the other
- Compare booleans to see if they are equal
- Combine boolean and relational operations to complete complex tasks
- Compare String data, to see if two strings are equal
- Construct if-else-if chains

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

11.1 **Introduction**

Boolean logic is at the heart of computer science. Computer have to look at data, and make decisions based on the circumstances. As such, boolean logic scales up very quickly beyond the simple examples we have seen, and can get very complicated. Think about it – all the software you use on a computer, phone, or tablet, is built using `if` statements. In this unit we look at the next level of difficulty with conditionals.

11.2 **Relational Operators**

There is only so much you can do with the built in booleans, and with our AND, OR, and NOT operation, so we need to generate new booleans that relate more to our own needs and problems. We do this by using relational operators: operations in Processing that result in a boolean value by comparing two values.

The simplest relational operator is to compare two values to see if they are equal. You can do this by using two equal signs together, for example, `number1 == number2`, gives a boolean result. ***Be careful! Use two equal signs stuck together, not one – one equal sign is for setting a variable to a value.***

For example, you can test if two numbers are the same:

```
int a = 5;
int b = 6;
println(a == b);
```

This will print true if `a` and `b` have the same value, and false if they do not. As we can see, here we will see `false` as our output, since a and b are not equal.

Just like our logical operators in the previous unit, this takes to values, compares them, and gives a boolean result.

For example, we can detect if the mouse is on the diagonal of the canvas, by seeing if its x and y coordinates are the same.

```
// brackets are not required but help readability
boolean onDiagonal = (mouseX==mouseY); // true if X equals Y
```

Let's make a drawing program to automatically draw a grey circles when the mouse hits the diagonal.

```
if (onDiagonal)
{
  stroke(127)
  ellipse(mouseX,mouseY,10,10);
}
```

So, in this program, each time it draws, it compares the mouse X and mouse Y. If they are the same, then `onDiagonal` is true, and false otherwise. Then, if they are the same, we draw a circle.

There are other relational operators, too:

!=	a != b	true if not equal
<	a < b	true if a is less than b
<=	a <= b	true if a is less than or equal to b
>	a > b	true if a is greater than b
>=	a >= b	true if a is greater than or equal to b

Be careful! The order of the <= and >= is important, for example, you cannot do => and =<.

In all of these cases, you can compare values and you will get a boolean result based on those values. In addition to storing the result into a variable, you can also use the boolean operators directly within an `if` statement. For example, above we had the following code.

```
boolean onDiagonal = (mouseX==mouseY); // true if X equals Y
if (onDiagonal)
{
  // ...
}
```

This can be simplified to:

```
if (mouseX == mouseY)
{
  // …
}
```

Which not only saves a variable and is shorter, but is also quite intuitive to read.

Advanced: You may see people using characters with the less than or greater than comparisons. The reason this works is because it looks at the numerical representation underneath (review in Unit 8.7). Some people use this to see if a character is a letter or number, for example, seeing if a character is within the range of A and Z, leveraging the structure of the ASCII table. This is generally considered bad practice, since it is not Unicode-safe, and introduces a whole slew of issues on international platforms (e.g., the web, or non-English machines). Don't do it.

When using the `==` operation, one thing you will see people doing is to compare boolean variables to true or false, for example:

```
if (onDiagonal == true)
```

Or

```
if (onDiagonal == false)
```

Logically, this works – you are comparing a boolean to true or false. This is not an error per se, but is generally frowned upon, since it is redundant. If you think about it, the *definition of an `if` statement* is to test if a boolean is true. By adding the `== true` in there, it's akin to saying "if a is true is true", which sounds silly. Experienced programmers notice this right away, and you shouldn't do it.

It can also introduce errors in annoying ways. Consider the following buggy code. Can you see the bug?

```
boolean b = true;
if (b = true)
{
  println("true!");
}
```

Here, the programmer used one `=` and not two `==.` So, instead of comparing `b` to `true` and giving a boolean result, the code *assigns* `b` to `true` (one `=` is used), and then uses the result, `b`, in the test. This if statement a) sets `b` to `true`, and b) always evaluates to `true`, and runs. Nasty bug.

You can avoid these bugs by not doing the `== true` and `== false` style. Instead, test if a boolean is true with `if (b)`, and test if it is false by using negation, `if (!b).`

There are some caveats to the relational operators, unfortunately. These relational operators usually work as you would expect. For example, they are very robust with integers, can be used with characters and booleans, but there are important exceptions that we need to cover. These are so important, that they get their own subsections.

11.3 **Comparing Strings**

If you remember, Strings are objects, and so have annoying quirks. For relational operators, the greater and less than variants clearly do not make sense for strings. However, how do you compare if two strings store the same text?

So far, we compare two pieces of data by using two equal signs: ==. Unfortunately, we cannot use this technique with strings. While it will seem to work (that is, it runs and does not crash), it does not do what you think: using == on Objects tells you if they are the same object in memory, an advanced concept we do not yet cover. Since we didn't learn objects yet, this gets very confusing and you shouldn't use == on strings. Expect bugs if you do.

We need to learn a new technique that lets us compare strings. If you remember, since Strings are objects, we can do extra things to the variable using methods. That is, we can run functions on the variable itself. We already learned some methods, e.g., to check how long a string is.

In this case, we have a method to check if a string has the same data as another string. This is the `.equals` method, which gives you a boolean. It is used as follows:

```
stringVariable.equals(otherString)
```

For example:

```
String s = "Jim";
if (s.equals("Jim")) // compare s to "Jim"
{
  println("yay!!!");
}
```

In the above example, the `s` variable data is compared to the literal string `"Jim"`. They are equal, so it gives a `true`, and the `if` statement block executes.

You can also compare two string variables this way:

```
String s1 = "cat";
String s2 = "dog";
println(s1.equals(s2));
```

Be careful, though, since string comparison is *case sensitive*. The following example gives you false:

```
String s1 = "jim";
String s2 = "Jim";
println(s1.equals(s2));
```

We won’t use string comparisons much in this course, but you should be aware of it. It is definitely testable material.

11.4 **Comparing Floating Point Numbers**

As we learned in Unit 7, computers cannot store perfect real numbers. Instead, they store approximations only. For the most part, this is okay, and we do not worry about it. However, when comparing floating point numbers, we need to be aware that we may have rounding errors based on how computers store the numbers.

Luckily, the way the numbers are stored is consistent, so the greater than and less than operators are robust for floating point numbers.

Unfortunately, due to precision error and rounding error, we cannot trust the == and != operators on floats. Pretty much ***you should never use*** == ***and != on floats.***

Consider the following example:

```
if (0.7 == 0.1+0.6)
{
  println("they are the same!");
}
```

What is the output? Nothing, unfortunately. The `if` statement evaluates to false due to floating point rounding error. You can take a closer look at this by using a few `println` statements to put the results to console.

So how then do you tell if two floating point numbers are the same? The answer lies in realizing that floating points are approximations at best. In the real world, if you are cutting a piece of wood to be 1m long, how precise do you need it? Do you care if it is 1.1m long? probably. Do you care if it is 1.001m long? (1 m and 1mm). Probably not. In this example, you have a tolerance for how close you are before you consider it to be 1m long, probably +- 2 or 3 mm.

We do the same with floating point numbers. You can take the difference of two, and see how close they are. You can then look at this closeness and make a judgment call based on your application.

11.5 **Example: Tiered coloring**

Let’s modify our drawing program from the last unit (with appropriate active processing template code):

```
if (mousePressed);
{
  line(pmouseX,pmouseY,mouseX,mouseY);
}
```

so that the drawing color changes depending on where you are drawing. Let’s make five bands – if you draw less than x of 100, use color 50. Between 100 and 199, color 100. 200-299, color 150, 300-399 color 200, and 400-500 color 255.

For simplicity, I am starting from the base case where you draw only if the mouse is pressed, and clear the screen on a key press.

One way to approach this is to start with the first case:

```
if(mouseX < 100)
{
  stroke(50);
}
```

Make sure to place this before the line code to ensure that the stroke is properly set before drawing (and not after!);

For the next condition, we could simply add this condition following the first one:

```
if (mouseX < 200)
{
  stroke(100);
}
```

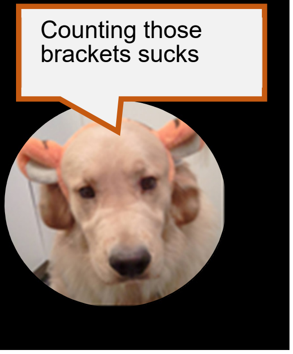
But there is a problem here! (try running it). This condition overrides the first case. For example, the coordinate 50 is true for both the above tests, and the color is always 100, and not 50.

One solution is to only test for < 200 if we know it’s not <100. The result will be a number from 100-199 (not < 100, but < 200). That is, if it’s NOT < 100, then test. We can do this using our `else` block and nesting an if inside there.

```
if (mouseX < 100)
{
  stroke(50);
}
else // not < 100, so >= 100
{
  if (mouseX < 200) // >= 100 but < 200
  {
    stroke(100);
  }
}
```

Let’s now look at the next case – 200-299. We follow the same logic as above. If our test for <200 fails, then it must be >=200, so we add an else block to THAT `if` statement. This gets confusing fast.

```
if (mouseX < 100)
{
  stroke(50);
}
else // not < 100, so >= 100
{
  if (mouseX < 200) // >= 100 and < 200
  {
    stroke(100);
  }
  else // not < 200, so >= 20
  {
    if (mouseX < 300) // >= 200 and < 300
    {
      stroke(150);
    }
  }
}
```



Whew! And all these ifs are nested inside the draw `block`. We have two more to go! This is getting very messy very fast (and those closing brackets and opening brackets are really confusing. There must be a better way!

In fact, there is. Because this if-else-if pattern is so common, we have special syntax for it.

11.6 **If-else-if chains**

When you have a series of `if`s, `else`s, and then nested ifs, you can use the following cleaner syntax that doesn’t nest ridiculously.

```
if (condition)
{
}
else if (condition) // only if above condition was false
{
}
```

```
else if (condition) // only check if all above are false
{
}// you can have as many of as you like
else // only run if ALL the above conditions
are false
{
}
```

Keep in mind that each `else if` is only checked if all the above ones were false. At any point, if any `if` test is true, then we run that block and no other conditions are checked – because we don't have an `else`!

The final else, called a terminal else, is only run if all the conditions above are false. ***The terminal else is optional, you do not need it.***

Let's rewrite the above partial solution using the new syntax:

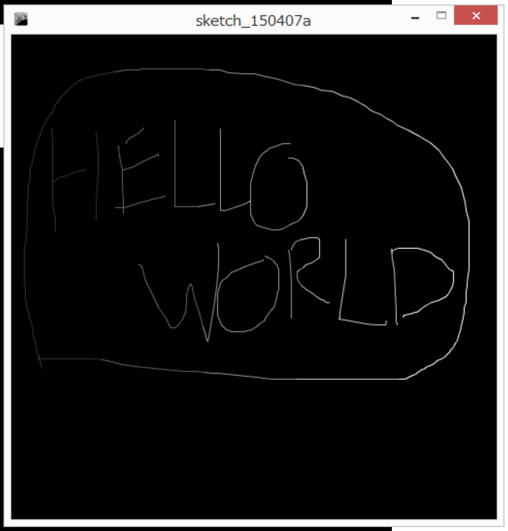
```
if (mouseX < 100)
{
  stroke(50);
}
else if (mouseX < 200) // >= 100 and < 200
{
  stroke(100);
}
else if (mouseX < 300) // >= 200 and < 300
{
  stroke(150);
}
```

This is much cleaner, and easier to read. Now we can easily add the next conditions. Finally, we can end with the final `else` for the last column, from 400-500:

```
if (mouseX < 100)
{
  stroke(50);
}
else if (mouseX < 200) // >= 100 and < 200
{
  stroke(100);
}
```

```
else if (mouseX < 300) // >= 200 and < 300
{
  stroke(150);
}
else if (mouseX < 400) // >= 300 and < 400
{
  stroke(200);
}
else // must be >= 400
{
  stroke(255);
}
```

Great! It works! Here is my final code for this program:



```
void setup()
{
  size(500, 500);
  background(0);
}

void draw()
{
  stroke(255);

  // only draw when pressing the mouse button
  if (mousePressed)
  {
    if (mouseX < 100)
    {
      stroke(50);
    }
    else if (mouseX < 200) // >= 100 and < 200
    {
      stroke(100);
    }
    else if (mouseX < 300) // >= 200 and < 300
    {
      stroke(150);
    }
    else if (mouseX < 400) // >= 300 and < 400
    {
      stroke(200);
    }
    else // must be >= 400
    {
      stroke(255);
    }

    line(pmouseX, pmouseY, mouseX, mouseY);
  }

  // erase the screen if any key is pressed
  if (keyPressed)
  {
    background(0);
  }
}
```

Here is a variant of the above if-else-if chain that has a bad bug: can you see it?

```
if (mouseX < 500)
{
      stroke(255);
}
else if (mouseX < 400)
{
      stroke(200);
}
else if (mouseX < 300)
{
      stroke(150);
}
else if (mouseX < 200)
{
      stroke(100);
}
else
```

```
{
      stroke(50);
}
```

The first test will always be true, since the mouse is always less than 500 on the X axis. Due to a concept called *short circuiting*, once a value is true, no other condition is checked. There is no searching for a best fit.

Remember: the computer works its way from top to bottom. If a condition is false, the next one is checked, until the end. If all conditions are false, the terminal `else` block runs, if it exists. From top to bottom, if any condition is true, then that block runs, and the rest of the chain is skipped.

11.7 **Example: Click on a Button**

Let's do a basic button. First, let's simply draw a square to act as our button. Set whatever background and drawing colors you like.

```
rect(100,100,50,50);
```

Let's make the rectangle change color if the mouse is over it. That is, we need some boolean tests to determine if the mouse is intersecting the rectangle, and based on that test, change the draw color. At this point it is a good idea to re-factor the code using good variable names that specify the button location and size. Declare them at the top of your program and initialize them, so that you can type:

```
rect(buttonX, buttonY, buttonSizeX, buttonSizeY);
```

These variables help us test the mouse position against the button position.

To solve this problem, let's tackle one piece at a time. Let's focus on X first. To hit the button, the `mouseX` has to be bigger than or equal to the button's left side. Let's setup the basic initial test, and the color change:

```
if (mouseX >= buttonX) // to the right of button left edge
{
  fill(127);
}
else
{
  fill(255);
}
```

Try running the code. The button changes if we are to the right of the button left edge. Of course, this also includes beyond the button right edge, which isn't right. So, let's add another condition. In regards to the X, we are on top of the button if we are >= the left edge, AND, we are <= the right edge. How do we calculate the right edge? `buttonX+buttonSizeX`:

```
if (mouseX >= buttonX  &&  mouseX <= buttonX+buttonSizeX)
```

So this works with respect to the X, now we need to also add the Y conditions. You can simply just keep on adding your &&s to have more AND conditions.

```
if (mouseX >= buttonX && mouseX <= buttonX+buttonSizeX &&
     mouseY >= buttonY && mouseY <= buttonY+buttonSizeY)
```

Now, the button changes color if we mouse over it! This is the calculation your computer performs every time a button changes color or highlights when the mouse goes over it. Try to add one more condition – only change color IF the mouse button is pressed in addition to the other conditions. Again, you can just chain these.

```
  if (mouseX >= buttonX && mouseX <= buttonX+buttonSizeX &&
      mouseY >= buttonY && mouseY <= buttonY+buttonSizeY &&
      mousePressed) // all four edges
```

We now have a button that we can press!

This logic gets confusing fast, particularly when you start mixing AND and OR and different relational operators. One thing you can do is to add brackets to help clarify it visually, although it is not necessary.

```
if ((mouseX >= buttonX) && (mouseX <= buttonX+buttonSizeX) &&
    (mouseY >= buttonY) && (mouseY <= buttonY+buttonSizeY) &&
     mousePressed) // all four edges
```

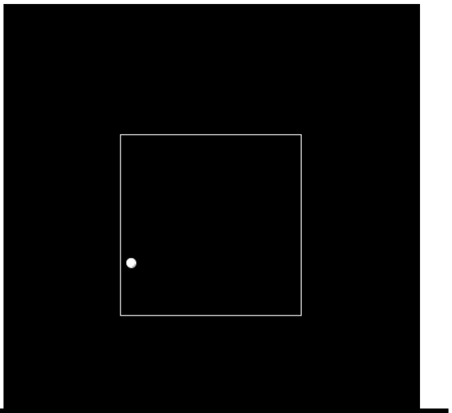
This small change can really make it easier to see.

Here is my whole program:

```
int buttonX = 100;
int buttonY = 100;
int buttonSizeX = 50;
int buttonSizeY = 50;
```

```
void setup()
{
  size(500, 500);
  background(0);
}
void draw()
{
if((mouseX >= buttonX) && (mouseX <= buttonX+buttonSizeX)&&
    (mouseY >= buttonY) && (mouseY <= buttonY+buttonSizeY) &&
     mousePressed) // all four edges
  {
    fill(50);
  }
  else
  {
    fill(255);
  }

  rect(100, 100, 50, 50);
}
```

11.8 **Example: Rebounding Ball**



Let's do a simple model of a ball bouncing around inside a box. Let's draw a box in the middle of the screen, and have a ball moving inside it. When the ball hits an edge, make it bounce.

First, let's make a ball start at the center of the screen, and give it an initial speed. We can set its speed in the x and y directions independently. Changing these numbers changes the angle it moves at. Here are my initial variables:

```
int ballX = 250; // start at the center of the screen
int ballY = 250;
int ballSpeedX = 2;
int ballSpeedY = 1;
int ballSize = 10;
```

In the draw loop, we need to draw the ball, and make the ball move by the speed amount each frame. Keep in mind that a negative speed just means that the ball moves in a different direction.

```
// draw the ball
ellipse(ballX, ballY, ballSize, ballSize);

// move the ball
ballX += ballSpeedX;
ballY += ballSpeedY;
```

Now, we need to draw the box that the ball bounces around in. I will setup some variables to define the box, and then draw the box in the draw loop:

```
int boxLeft = 150;
int boxTop = 150;
int boxSize = 200;
int boxRight = boxLeft + boxSize;
int boxBottom = boxTop + boxSize;
...
// draw the box
stroke(255);
line(boxLeft,boxTop,boxRight,boxTop);
line(boxRight,boxTop,boxRight,boxBottom);
line(boxRight,boxBottom,boxLeft,boxBottom);
line(boxLeft,boxBottom,boxLeft,boxTop);
```

Now you should see a ball that starts in the center of the screen and moves inside a box. Of course, it just goes through the box!! We need to make the ball bounce.

If you remember your physics, there are real formulas to calculate how a ball bounces, but that is much too complex for us here. All that we need to do to simulate a reasonable bounce is: if the ball hits the side walls, change the direction of the X movement. If it hits the top or bottom walls, change the direction of the Y movement. We can change these directions simply by multiplying by -1.

```
// check the bounds
// right and left
if (ballX < boxLeft || ballX > boxRight)
{
  ballSpeedX *= -1;
}
```

```
// top and bottomm
if (ballY < boxTop || ballY > boxBottom)
{
  ballSpeedY *= -1;
}
```

That's it! We now have a bouncing ball inside a box. I encourage you to play with this example, such as doing things on a key or mouse press, moving the ball around, or changing the speed.

11.9 **Logic Practice and Short Circuiting**

Logic can nest and get complex very quickly! Here are some examples. Look at the following expression:

```
!( !(a<b) || c)
```

Fast: what are the data types of a, b, c? Can you tell?

`a` and `b` must be numerical, since you are comparing if one is less than another. `c` must be a boolean since it is being used in an OR operation.

What is the result if `a=3`, `b=1`, and `c=true`?

Let's work through it

```
!( !(a<b) || c)
!( !(3<1) || T)
!( !(F) || T)
!( T || T)
!( T)
F
```

In general you want to avoid such annoying and complex statements in your programs. If you find yourself making these kinds of messy expressions, you can fix it with better variable names, and doing some of the calculation first into a variable to help the reader understand what is happening.

In the above example, do you see a shortcut to get the final F answer without doing all the calculations? In fact once you know that `c==true`, you do not need to do the rest of the test at all. If you have one true value in an OR (||) statement, then the OR is true, so we know that !(a<b) || c is true if c is true, regardless of the values of a and b. This is general idea is called short circuiting and you should be aware of it. In short,

the computer can detect a short circuit, and some commands may never be run. For example

```
if (myVariable && someCommand())
```

In this case, if myVariable is false, then the whole and statement is necessarily false, and the computer may not run someCommand. That command may have some important side effects (like drawing something) that you expect to happen (although this is a terrible idea and you should never do that!). At this point, you will not likely run into this problem, but it is important to be aware of.

Advanced: You can force your computer to not short circuit a logical operation. There are special commands for this. In Processing, instead of && and || for and and or, you can use the single versions, & and |. These behave the same on boolean values as the regular and and or, without short circuiting. Most people are not aware of these, and they are not commonly used.

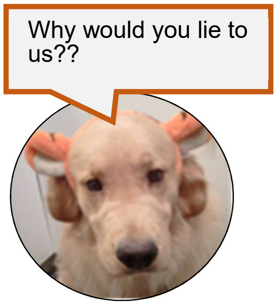
11.10 **Alternate If Statement Syntax**

We learned that the if statement requires a code block to run, in the case that your test is true:

```
if (test)
{ // code block
}
else
{ // else block
}
```

Actually, these blocks are not always necessary. In the special case where you only have one command to run if a statement is true or false, then you can avoid the block and just put your single command:

```
if (test)
  oneCommand();
else
  otherCommand();
```



Although this may seem like a great way to save space, it generally is a bad idea because it can easily lead to problems. Consider the following example:

```
if (shouldMugJim)
    putOnBlackMask();
    practiceToughVoice();
    mugJim();
continueProgram();
```

Even though the indentation is setup to look like all three commands will be run if the if test is true, because code blocks aren’t used, only the first command (`putOnBlackMask()`) is run. The other two run anyway, regardless of the outcome of the test. That is, if `shouldMugJim` is false, you still practice your tough voice and mug Jim, without putting a mask on. To make things worse, when you do if-else-if chains, you can mix and match, as follows:

```
if (booleanStatement)
{
     commandA1;
     commandA2;
}
else if (booleanStatement)
     command B;
else
     command C;
```

This creates even more opportunities for bugs. For example, can you spot the bug in the following code?

```
if (temperature < -50 || temperature > 50) {
// invalid temperature
     severity = -1;
} else { // temperature is valid
  if (temperature < -20)
   severity = 4;
  } else if (temperature < -10) {
   severity = 3;
  } else if (temperature < 0) {
   severity = 2;
  } else
     severity = 1;
}
```

In this case, there is a missing open bracket after the test for `temperature < -20`, so that `if` test does not use a code block. When the closing bracket is encountered after `severity = 4`, it is closing the else block for temperature is valid. Then, we are trying to follow a final terminal else with a new else if, which doesn't make any sense. This won't compile. Think through this one.

ALWAYS using explicit code blocks avoids this oversight risk:

```
if (shouldMugJim)
{
    putOnBlackMask();
    practiceToughVoice();
    mugJim();
}
continueProgram();
```

Check your Understanding

11.11 Check your Understanding: Exercises

Exercise 1. Make a basic drawing program as in this chapter, with a black background and white drawing ink, which draws while the user is pressing a mouse button.

a. Update the program so that it only draws if the mouse is in the top left quadrant of the screen.
b. Change the program so that it only draws if the mouse is in the top right quadrant of the screen.
c. Change the program so that it only draws if the mouse is NOT in the bottom right quadrant (so it draws in the other three).

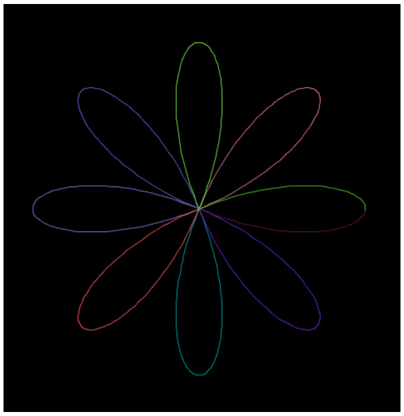
Exercise 2. Update the rebounding ball example (11.8) so that the ball bounces when the *edge* of the ball hits. Currently, it bounces when the center hits a wall.

Exercise 3. Make a simple drawing program that has four buttons on the top of the screen.

a. If the first button is pressed, clear the canvas
b. Second button: set the color to white
c. Third button: set the color to grey
d. Fourth button: set the color to black (erase)

Exercise 4. Update the simple button clicking example in 11.7, to turn the button into a toggle button. If you click it once, it stays down. If you click again, it comes up. You will need global variables to remember the state, and, a technique (as in the previous unit) to ensure that only one change happens when the mouse button is pressed (and that it doesn't rapidly toggle).

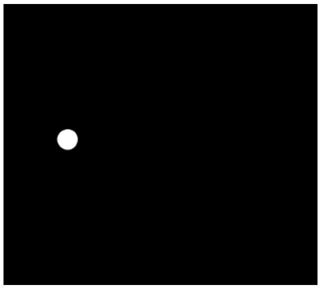
Exercise 5. Update Exercise 5 in section 7.8, the rose program. (hopefully, you saved your exercise from earlier!). Now, make it so that the rose has a random color on each petal.



a. Determine whether the current point is farther from the center than the previous one. Use a **boolean** variable to store this information. That tells you whether the line is moving away from the middle, or toward it.
b. When do you change the color? When the line is now moving away from the center, but it wasn't last time. You will need a **boolean** expression to figure this out.
c. Now you need to keep more information from the last frame: How far away the last point was, and whether it was moving toward the center, or away from it. Use global variables again, one of them a **boolean** variable.
d. To pick a random color you can use **stroke(random(256), random(256), random(256));**

Exercise 6. This unit introduced a problem of a ball bouncing around *inside* a square (11.8). Instead, make the ball bounce around outside the square. That is, it bounces off the walls of the canvas (like inside a square), but additionally, it stays outside of a square placed in the middle of the canvas. This is deceptively challenging, and a lot harder than it looks.

Exercise 7. Mke a version of the game "Whack-a-Mole", where you have to hit targets with your mouse before they disappear. This will be challenging, with quite a bit of logic, and many things to keep track of, using global variables. The game works as follows. A "mole" gets generated at a random location. It starts as a circle of size 0 and then grows at some speed until a maximum size is reached (use global named constants, of course). Then it starts shrinking. When it reaches size 0, a new mole starts to grow at a new random location. But if the user clicks on a mole before it disappears, the mole is instantly "killed", the player's score increases by one, and a new mole begins somewhere else. Hint: Use the



Quick! Click on the circle before it disappears!

distance from the mouse to the center of the mole to detect a “hit”. After this much of the game is working, you can make it fancier. Make the moles different colors or sizes or speeds. Give more points for smaller or faster ones. Make the moles go faster the longer the game lasts. Add a “game over”. Print the score in the window (look up the “text” command). Have fun with it!

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 12. LOOPS

Summary

You will look at how to make Processing repeat a command a given number of times, or, while some condition is true. This greatly increases the power of your programs. In this section, you will…

- Learn about two different kinds of loops, for making the computer repeat operations: the “for” loop and the “while” loop
- See how loops impact variable scope
- Learn about common problems with loops
- Get a great deal of practice with loops.

Learning Objectives

After finishing this unit, you will be able to …

- Create a `for` loop to repeat an operation a fixed number of times.
- Create a `while` loop to repeat an operation while some condition is true.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

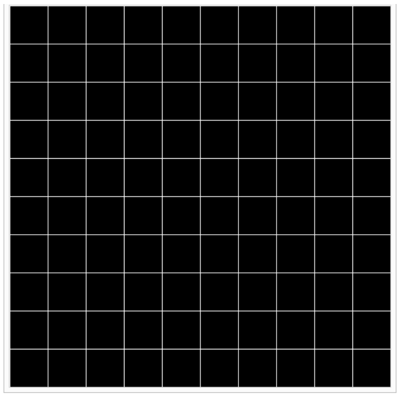
12.1 **Introduction**

So far in our programs, we have to explain every single operation that we want the computer to do. Using `if` statements some of our programming can be conditional based on the circumstance, but for things that are done, we give each command in an operation. It would be nice if we could provide more general commands to the computer, to more easily do more work.

Loops are simple at their core, but for many, are a new way of thinking. As such, we have included a large number of examples in this chapter.

12.2 **"for" loops**

For example, let's draw horizontal and vertical lines to make a grid. How would we do this with the tools we have so far?



First, let's establish the dimensions of the grid, and use that to calculate the height and width of the cells.

```
int grid = 10; // 10 by 10 grid
int cellSizeX = width/grid;
int cellSizeY = height/grid;
```

Now we can start drawing our grid. Let's draw the horizontal lines first. Each line is basically offset a multiple of the cell size.

```
line(0,0,width-1, 0); // top line
line(0, cellSizeY*1, width-1, cellSizeY*1);
line(0, cellSizeY*2, width-1, cellSizeY*2);
```

Try this out. As you can see, we are drawing horizontal lines, and jumping down the cell height for each line. For 10 cells, we need 11 of these line commands (not 10!). We need 11 additional commands for the vertical lines, for a total of 22 line commands. What if we wanted a 100x100 grid? That is 202 commands! Clearly we can't type this in by hand, and even using copy paste, this is very tedious.

Wouldn't it be nice if we could re-factor the above code using something like the following...

- For every value of in the range of 0..10
- Do a line from 0,cellSizeY*value, to width-1,cellsizeY*value

Think about it, and compare this to the three lines above. If we could tell the computer to repeat that line command with different values multiplied by our `cellSizeY`, we'd save a lot of work over typing them all up. As the value increases, our line moves down the screen. Luckily, we can do this. Let's learn Processing syntax to do this. This is called a "for" loop.

```
for (initial command; end condition; upkeep command)
{
  // code block
}
```

When Processing encounters the code, it does the following operations: ***Memorizing this sequence is immensely important, do it!***

1. The `initial command` runs one time, at the start
2. The `end condition`, a boolean, is checked. If it is `true`, we run the `code block`
3. The `code block` runs like normal processing code
4. The `upkeep command` is run
5. Go to step #2

Up until now, everything in this list follows what you have learned except for step 5. Now, the program can go back, and *repeat* something again and again, until the `end condition` is false. This is why is called a loop, as the code block runs again and again.

It's useful to think of this code in terms of what we already understand. Let's work from the following *fake* code example.

```
int i = 0;
if (i < 10)
{
  // draw line at grid position i
  line(0, cellSizeY*i, width-1, cellSizeY*i);
  i++;
}  // go back to the if and do it again
```

Here, we create a variable and give it an initial value. We use an `if` test to check for some end condition, for example, if `i` is less than 10. Then we run the code block. At the end of the code block, we make `i` larger by one. Then, it would be nice to go back to the `if` and do it again.

You can see that every time we re-do the `if` statement, `i` increases, until `i<10` is false – that is, `i` is `10` or larger. This probably looks confusing, so let's have a concrete example using the actual syntax of a `for` loop. The following template of a `for` loop is by far the most common use of the loop. Let's assume that `times` is set to 4.

```
for (int i=0; i < times; i++)
{
  // code
}
```

Let’s look at this using the order of operations specified on the previous page.

1. The `initial command` runs one time, `int i = 0;`
2. The `end condition` is checked. if `i<times`, then we run the code, otherwise, we quit the loop.
3. `i` is 0, so `i<times`, or `0<4`, is true. The `code block` is run
4. The `upkeep command` is run, `i++`
5. Go to step #2 and repeat

If `times` is set to 4, then the loop will run 4 times: once with `i=0`, `i=1`, `i=2`, and `i=3`. When `i` is 4, then `i<times` is false (`4<4` is false), and the loop ends.

Note: This is a key spot for the dreaded off by one error. Since our loop starts at 0, if we want it to run n times, the final run is with n-1.

You can put any valid Processing code in the initial condition, end condition, and upkeep, but it makes most sense usually to stick to the above template. Let’s re-consider our grid line example. Here are some of the line commands:

```
line(0, 0, width-1, 0); // top line
line(0, cellSizeY*1, width-1, cellSizeY*1);
line(0, cellSizeY*2, width-1, cellSizeY*2);
```

Looking for the pattern here, we can factor out the `cellSizeY*1,2,3`… and make it `cellSizeY*i`:

```
line(0, cellSize*i, width-1, cellSize*i);
```

If we complete the above command for the values of `i` from 0 to 9, we get 10 lines. Let’s wrap it in a `for` loop to make `i` go from 0 to the grid size:

```
for (int i = 0; i < grid; i++)
{
  line(0, cellSizeY*i, width-1, cellSizeY*i);
}
```

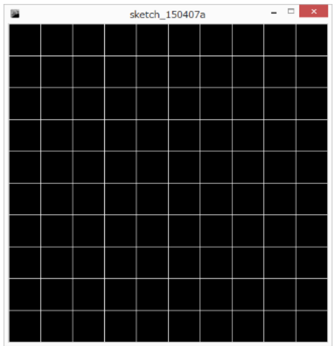
At the beginning, `i` is 0, so we get our first line command. Then, `i` increases to 1, and we get the second line command above... this continues until `i` = 9, and we get the 10th command. When `i` is 10 (grid’s value), our test is false, and the loop ends.

This gives us 10 lines. Above we said we wanted 11 lines – since for *n* cells there are *n+1* lines. Try updating the loop to give that last line (there are several ways to do this).

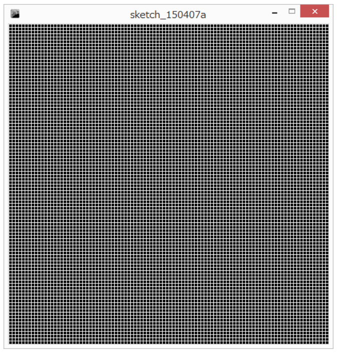
You can modify the above loop to add the vertical lines as well. A second loop is not needed. The calculation for the vertical lines is very similar to the horizontal ones:

```
line(cellSizeX*i, 0, cellSizeX*i, height-1);
```

You have the following grid! Wasn't that easier than typing all those lines manually? If you can think of your problem more generally, you can use the power of `for` loops to save yourself a bunch of work.

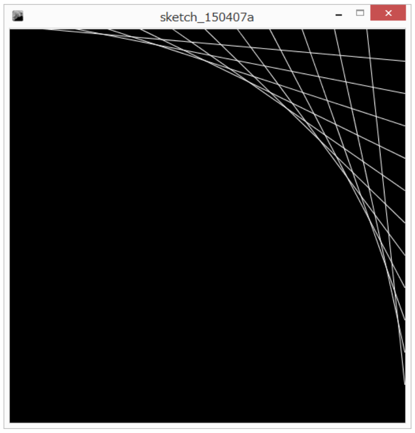


Now, let's look at another great power of loops. If you setup things generally, like we did here, things scale up very easily! With this code, you can make a grid of any dimensions just by changing that value of the grid variable. Change that variable, and all your calculations change, including how many times your `for` loop runs. Change the grid to 100 and you get the following result, without any additional work!



12.3 **Example: Line Art**

You may remember in elementary school making some line art like the image to the right. You take a ruler and draw straight lines between the first point along the top, and the first point on the right. Then you draw between the second point on the top and the second on the right. You do this for all the points, and you get an approximated curve. The denser your points are, the better your curve is approximated. We can now do this example using `for` loops. In fact, this is very similar to our grid example.



First, we need to calculate how many points across the top we want, and the spacing of the point. This is identical to our grid above. Then, we use a `for` loop to iterate through all the points and then draw lines between them.

To start something like this, first think about the line you want to draw before you think of the loop. If you do a couple of examples, then it usually helps you to see a pattern that will work in the loop. Assuming that we have the following variables:

```
int grid = 10;
int cellSizeX = width/grid;
int cellSizeY = width/grid;
```

Then we can draw some lines. The code for the first three lines would look as below, assuming we draw from the top of the screen to the right of the screen. We would draw all the way along, which each line being FROM (x=cell, y=0), TO (x=width-1, y=cell). For example, the first line would be from the 1st cell along the top left, to the 1st cell down the right, the second line would be from the 2nd cell along the top left, the 2nd cell down the right, etc.

```
line(cellSizeX*0, 0, width-1, cellSizeY*0);
line(cellSizeX*1, 0, width-1, cellSizeY*1);
line(cellSizeX*2, 0, width-1, cellSizeY*2);
```

From here, you can see that the general case would be rewritten as

```
line(cellSizeX*i, 0, width-1, cellSizeY*i);
```

for cell `i`. Now, we wrap this in a for loop that goes through all the cells

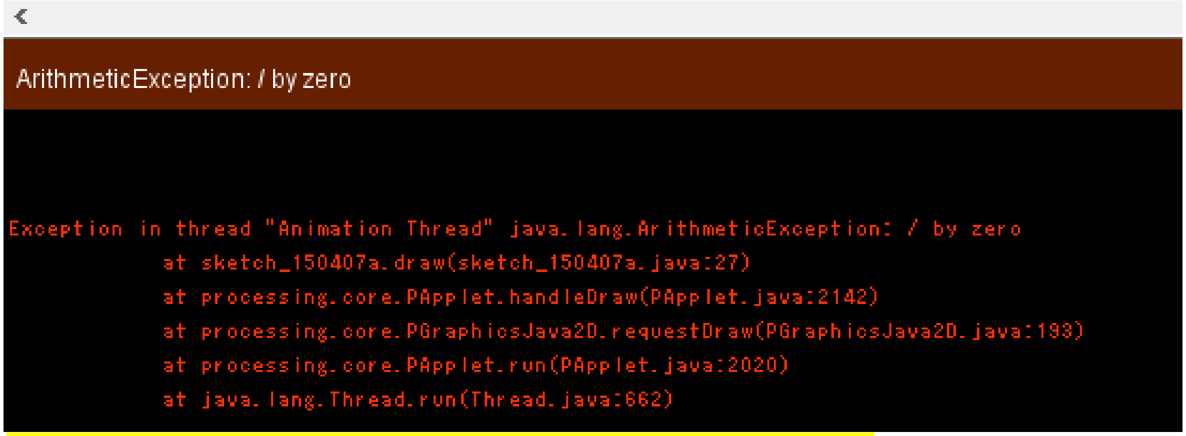
```
for (int i=0; i<grid; i++)
{
  line(cellSizeX*i, 0, width-1, cellSizeY*i);
}
```

And it works! Make sure your code is up and running at this point.

To make this interactive, try setting the grid at the beginning of the draw block using the mouse coordinate. What I did was to set the grid = mouse/10, so that grid ranges from 1 to 49. Try this out to see what happens. Did it work okay? Did you try putting the mouse ALL the way to the left of the screen?

If you put the mouse all the way to the left of the window, the program crashes!! This may be your very first actual crash. If you look in the console of processing, it gives you a very very long description of what happened. Look at the image on the next page. If you look closely, there is some help in here. The first line helps you understand what happened, if you read between the cruft. It says you have a “`/ by zero`”. What happens if you divide a number by zero? It is undefined, impossible to do, so the computer panics and stops your program.

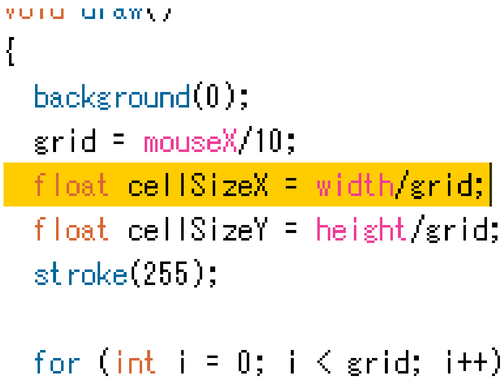
NOTE: this is called a run-time error. The program starts up but hits an error that it could not foresee ahead of time. This error makes the program stop



ArithmeticException: / by zero

```
Exception in thread "Animation Thread" java.lang.ArithmeticException: / by zero
	at sketch_150407a.draw(sketch_150407a.java:27)
	at processing.core.PApplet.handleDraw(PApplet.java:2142)
	at processing.core.PGraphicsJava2D.requestDraw(PGraphicsJava2D.java:193)
	at processing.core.PApplet.run(PApplet.java:2020)
	at java.lang.Thread.run(Thread.java:662)
```

running and it gives you some helpful (hah!) information.



```
void draw()
{
  background(0);
  grid = mouseX/10;
  float cellSizeX = width/grid;
  float cellSizeY = height/grid;
  stroke(255);

  for (int i = 0; i < grid; i++)
```

Also, notice that it tells you which line in your program. In this case, my program is the `sketch_numbers` program, so it tells me that my error was at line 27. Processing also highlights the line: As you can see here, the error happens when I try to divide by grid. Why would this be a divide by zero? When would grid be zero?

Aha! When `mouseX` is zero, grid is 0/10 which is 0. How can you fix this?

There are many ways. A simple way here is to simply add 1 to grid. We know that the very smallest possible value of `mouseX` is 0, so if we add one, we should be safe. The program now works. Here is my final program.

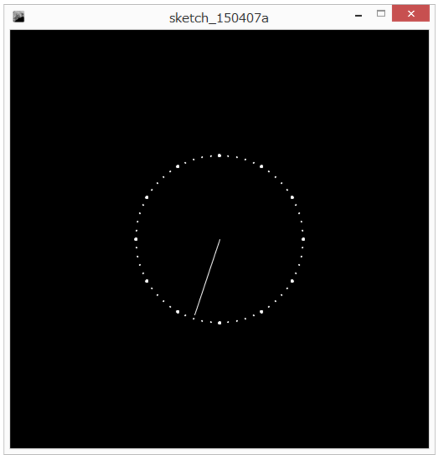
```
int grid = 10;
void setup()
{
  size(500, 500);
  stroke(255);
}

void draw()
{
  background(0);

  // calculate grid spacing
  grid = mouseX/10+1; // +1 to avoid / by 0
  int cellSizeX = width/grid;
  int cellSizeY = height/grid;

  // draw the lines
  for (int i = 0; i < grid; i++)
  {
    line(cellSizeX*i, 0, 499, cellSizeY*i);
  }
}
```

Notice how sometimes when you move the mouse some additional lines get drawn but the grid does not change? This happens because of the integer math that we are doing. When we do our divisions to calculate the cell sizes, as our `mouseX` gets larger, we have more of a rounding effect as the remainder is thrown away. Change all the math to floating point to get a smoother animation, as you use real fractions and do not throw away anything.

Note: why are we okay with using a single letter, `i`***, for our variable name?*** In general, single-letter variables are a bad idea. However, you can use them when it's convention, or follows a known formula. In this case, it is an accepted convention in both computer science and mathematics to use `i` for an *iterator*, a value that goes over a range of integers. Other common iterator variables are j and k.

12.4 **Example: Clock**

Let's make an analog clock. To keep things simple, let's just make it with the second hand, as in the inset. This will be one of our most complicated examples so far, and we'll see how many of the skills we learned so far come together to help us solve the problems.



The first problem is how to make the circle of dots around the clock. We need 60 dots, so doing this manually with a bunch of `ellipse` commands is probably a bad idea. We can use a `for` loop!

To approach this, let's consider an arbitrary dot at some angle θ. Remember from section 7.3 that, given an angle, we can calculate the x and y using sin and cos:

```
x= sin(radians)*radius

y= cos(radians)*radius
```

The challenge here is, how do we calculate all the angles for those dots? If we can get the angles, then we can get the x and y coordinates. The trick, which is very common with `for` loops, is to work in percentages. First, for a given dot we figure out what percentage around the clock we are, for that dot. Then, multiply that percentage by 2PI (a whole circle), to get radians. Given a dot `i` (we have 60)

```
float angle = i/60.0 * 2 * PI; // angle to dot
```

`i/60.0` gives the percent of the way around the clock. Don't forget to use 60.0. If `i` is an integer, what happens if we use just 60? Integer division! Now that we have the angle, we can calculate the x and y position. Let's start writing the program.

First, setup some great globals that describe your clock

```
int clockRadius = 100; // size of clock
int tickSize = 1; // radius of tick marks
```

Then, in the draw loop, use a `for` loop to go through the 60 tick marks, calculate their angles, and draw tick marks there. In this case, let's put the clock at the center of the screen.

```
// draw tick marks
for (int i = 0; i < 60; i++)
{
  float angle = i/60.0 * 2 * PI; // angle to dot
  // center these on canvas
  float x = cos(angle)*clockRadius + width/2;
  float y = sin(angle)*clockRadius + height/2;
  ellipse(x,y,tickSize,tickSize);
}
```

Try this, and make sure it runs. Do you recall the exact order of operations with the `for` loop? Does our loop run exactly 60 times? Or is it 59? 61?

There is one more thing. If you look at my example on the previous page, you can see that every 5th tick mark is actually quite a bit bigger, just like on a real clock. How can we go about doing this?

An easy way to do this – and one that is very common with computer scientists – is to use the modulo operator. Remember that we did something similar earlier. If you take some number *n* and take it modulo *m*, when do you get zero?

This is probably clearer if we use real numbers. If we take *n* modulo 5, when do we get zero? Zero means no remainder. There is no remainder when *n* is a multiple of five. For example, `0, 5, 10, 15, 20,` … We can use this to detect every fifth dot – when we take the number mod 5, and get 0, let's draw a bigger dot. This is a simple `if` statement, and I created a new global to define the large tick size:

```
if (i%5 == 0)
{
     ellipse(x,y,largeTickSize,largeTickSize);
} else {
     ellipse(x,y,tickSize,tickSize);
}
```

The last piece of the puzzle is how to get the second hand in there. For this, we need a new processing command:

```
int second(); // returns the current seconds from the clock
```

To draw the second hand, we need to first calculate the angle to use. This is very similar to above. We simply take this number, find the corresponding angle, and draw a line to the x and y point. Try it on your own before reading the following code. This doesn't belong inside the `for` loop, because we only draw one second hand.

```
angle = second() / 60.0 * PI * 2; // percent around circle
x = cos(angle)*clockRadius + width/2;
y = cos(angle)*clockRadius + width/2;
line(width/2, height/2, x, y);
```

Here is my final code:

```
int clockRadius = 100;
int tickSize = 1;
int largeTickSize = 3;

void setup()
{
  size(500, 500);
  stroke(255);
}

void draw()
{
  background(0);
  float angle;
  float x;
  float y;

  // draw tick marks
  for (int i = 0; i < 60; i++)
  {
    angle = i/60.0 * 2 * PI; // angle to dot
    // center these on canvas
    x = cos(angle)*clockRadius + width/2;
    y = sin(angle)*clockRadius + height/2;
    if (i%5 == 0)
    {
      ellipse(x, y, largeTickSize, largeTickSize)
    } else {
      ellipse(x, y, tickSize, tickSize);
    }
  }

  // calculate and draw the second hand
  angle = second()/60.0*PI*2;
  x = cos(angle)*clockRadius + width/2;
  y = sin(angle)*clockRadius + height/2;
  line(width/2, height/2, x, y);
}
```

12.5 Example: Strings

Let’s make a quick example to loop through a string, and put out each character at a given spacing. Let’s first setup our globals:

```
final int SPACE = 10;
final String message = "howdy!";
```

then in our `draw` loop, let’s use a `for` loop to go through each index of the string. Pay very careful attention to the `for` loop condition here. We loop `i` from index 0, to the last index, which is length -1. Therefore, we go while index is less than the length. Look at the following three, only the last one is correct

```
for (int i=0; i<=message.length(); i++) // 0..length
for (int i=0; i<message.length()-1; i++) // 0..length-2
for (int i = 0; i < message.length(); i++) // 0..length-1
```

at each index, then we grab the character in the string into a variable, and put it out at our given spacing. Luckily, the text command can take a character. You could also explicitly convert it to a string using the `str()` command. Let’s place the string at the `mouseY` just for fun. Inside the loop:

```
char c = message.charAt(i);
text(c, SPACE*i, mouseY);
```

Try increasing the spacing to 20 to see what happens. Next, let’s animate the spacing. To do this, we need to change the spacing each time we draw. To keep track of this, we need a global that can change (not a final!) to keep track of where we are. Let’s also define a range of spacing:

```
//final int SPACE = 10;
final int SPACE_MIN = 10;
final int SPACE_MAX = 100;
int space = SPACE_MIN;
```

Now, in our `draw` loop, we can simply increase our space by one each time we draw. If our space is larger than the max, re-set it to the minimum. You can try this yourself. Here is my final code:

```
final int SPACE_MIN = 10;
final int SPACE_MAX = 100;
final String message = "howdy!";
int space = SPACE_MIN;

void setup()
{
  size(500,500);
}

void draw()
{
  background(0);
  space = space + 1;
  if (space > SPACE_MAX)
    space = SPACE_MIN;
  for (int i = 0; i < message.length(); i++)
  {
    char c = message.charAt(i);
    text(c, space*i, mouseY);
  }
}
```

You can try touching this up, for example, to make the animation go backward when it hits the limit instead of starting over, or animate the y spacing as well.

12.6 **“while” loops**

There is another kind of common loop, called the `while` loop. The `while` loop, fundamentally, is simpler than the `for` loop, so this should not be too challenging.

for loops are useful when we know how many times to do something, as in our above examples. On the contrary, ***while loops are used when it is more difficult to determine how many iterations are needed***. In all of the above examples, we were easily able to determine or calculate ahead of time, for example, how many grid lines or clock dots we needed.

When we do a loop, we cannot always know how many times it needs to run. As such, `while` loops are simply setup with a block and an end condition, and does not specify how many iterations. Here is the basic syntax.

```
while (condition)
{
  // code block
}
```

You can think of this as basically an `if` statement that keeps repeating until it is false. In this case, Processing checks the condition first, and if it is true, it executes the code block. Then, it checks the condition again, and if true, executes the code block. Repeat *while* the condition is true, and stop when the condition is false.

As an illustrative example, let’s look at how to use a `while` loop to count to 1000. (You’d normally use a `for` loop). We need a counter variable, say `i`. At the beginning of the loop, we see if the variable is less than or equal to our maximum. Inside the loop, print out the value. Before iterating again, increase our counter:

```
int i = 1;
while (i <= 1000)
{
  println(i);
  i++;
}
```

Here, the while loop acts like an `if` statement: if it's true, do the block, if not done. The difference is, that once the block is finished, the `while` repeats by checking the condition again.

The order that things happen here is crucial. When the while loop starts, it first checks the condition. Since the condition is true, it executes the block. At the end of the block, it checks the condition again. If true, it executes, and so on. Repeat!

Important: the while loop only re-checks the condition at the end, and not mid-way. If your condition becomes false part way through the block, the block gets finished before checking the value again. The computer takes a really narrow vision and only looks at one operation at a time.

Just like for loops, while loops generally have four important parts. The `initial command` happens before the loop starts (`int i=1`). The `end conditon` is checked at the beginning of each loop (`i<1000`). The `code block` is run each time (`println(i)`), and the `upkeep command` is done at the end of the loop (`i++`).

In this case, you can see that the `while` loop and `for` loop are very similar. In fact, the above example can be written as

```
int i = 1;
while (i <= 1000)
{
  println(i);
  i++;
}
```

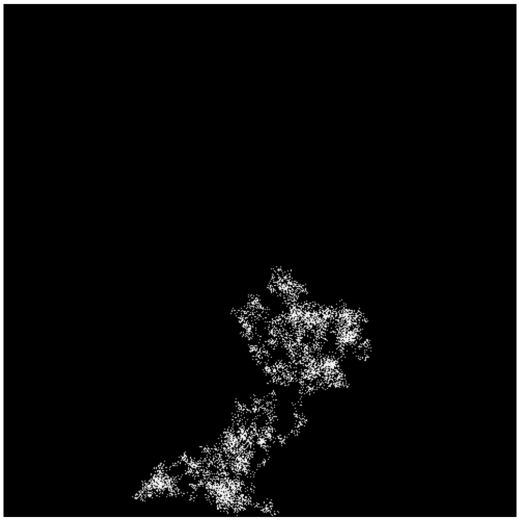
```
for (int i=1; i<=1000; i++)
{
  println(i);
}
```

And you get the exact same result.

Fundamentally, while `loops` are the basic loop and the `for` loop is just a syntax for a common usage of while loops. Actually, you can make any loop use one or the other, but it often makes more sense or is more intuitive to use a `for` loop or a `while` loop in a given scenario. Overall, `for` loops are great when you know ahead of time how many times the loop should run. `While` loops are better when you are unsure and cannot easily quantify the number of iterations. Now let's do some examples where `while` loops make more sense.

12.7 **Example: Random Walk**

Here is a simple example. Let's make a point start at the mouse, and move randomly about, and see what kind of path it takes to the edge of the screen.



Without loops, a random walk has to be animated: we do a single movement each time we draw, and then watch as the point moves around and finally goes off the screen. This can take a long time. Instead, we want to do this all at once, in one frame. We want to see immediately what path the point will take, as in the image to the right. To do this, we loop through random movements until the ball goes off the edge of the screen. If you want to use a `for` loop to do this, the problem is that you don't know how many times the loop will need to run, as it will depend on both where the point starts, and, what random moves it takes. Theoretically, it may go in circles for ever and never leave the screen, but it probably will ☺.

We can use a while loop to do this. Here is my overview of the draw function:

- Place a point at the mouse position
- Clear the screen
- While the point is still on screen
    - Do a random movement
    - Draw a point to represent the point
- Only do any of the above if the mouse is pressed (helps see the results better!)

Remember, this all happens each time you draw, before it gets shown on the screen. As such, it may actually take some time before you see the result.

Most of this code in the `draw` loop is straight forward. Here is a sketch:

```
void draw()
{
  if (mousePressed)
  {
    background(0);
    stroke(255);
    pointX = mouseX;
    pointY = mouseY;
    // while on screen?
    {
      pointX += (int)(random(MAX_MOVE*2+1))-MAX_MOVE;
      pointY += (int)(random(MAX_MOVE*2+1))-MAX_MOVE;
      point(pointX, pointY);
    }
  }
}
```

This way, every single time you draw, you generate a point at the mouse position, and loop random movements until it goes off the screen. You won’t see this animate because it all happens entirely in one draw.

The only part missing is the “while on screen” part. We can setup a boolean easily for this, and plug it into the while loop syntax above:

```
boolean onScreen = (pointX >= 0 && pointX < width) &&
                    (pointY >= 0 && pointY < height);
while (onScreen)
{
  ...
```

Here, we calculate a nice boolean and then plug it into the while loop. We could actually put the entire boolean into the while loop, but this longer approach increases readability, AND, gives a talking point. Let’s look at the whole draw loop now:

```
void draw()
{
  if (mousePressed)
  {
    background(0);
    stroke(255);
    pointX = mouseX;
    pointY = mouseY;
    boolean onScreen = (pointX >= 0 && pointX < width) &&
      (pointY >= 0 && pointY < height);

    while (onScreen)
    {
      pointX += (int)(random(MAX_MOVE*2+1))-MAX_MOVE;
      pointY += (int)(random(MAX_MOVE*2+1))-MAX_MOVE;
      point(pointX, pointY);
    }
  }
}
```

If you run this, the program doesn't display anything! What is happening??? Hint: when does the while loop end?

We aren't updating the while loop condition: the `onScreen` variable is calculated only once, before starting the while loop. At this point, it is using the point's starting position. Once the code is in the while loop, the `onScreen` boolean is never updated again. It always says true, and we have an infinite loop.

We need to update the `onScreen` variable inside the while loop, after we move the ball, so that it keeps checking. It should work now.

```
boolean onScreen = (pointX >= 0 && pointX < width) &&
                   (pointY >= 0 && pointY < height);
while (onScreen)
{
  pointX += (int)(random(MAX_MOVE*2+1))-MAX_MOVE;
  pointY += (int)(random(MAX_MOVE*2+1))-MAX_MOVE;
  point(pointX, pointY);
  onScreen = (pointX >= 0 && pointX < width) &&
    (pointX >= 0 && pointY < height);
}
```

Again, look at the above code. The following happens:

- The boolean `onScreen` is created and initialized based on the point's starting position
- The while loop starts, and checks `onScreen`. If true, it executes the block
  - At the end of the block, the `onScreen` variable is re-calculated
- Back to the beginning of the while loop, check again.

This way, `onScreen` is always up to date before being checked again in the while loop.

NOTE: the initial condition of this loop (just before the while), and the upkeep, are very similar (the same!). This is very common. Also note that the test is directly related to the upkeep – since the test should eventually be false.

12.8 **Example: count digits in a number**

If you are drawing a number on screen, it can help to know how many digits there are. For example, a number with more digits either requires more space, or should use smaller letters, than a number with fewer digits. We will make a program where the font size shrinks as the number gets larger.

Given a number *n*, we will calculate how many `digits` are in *n*, and then set the text size to `1.5*width/digits`. Then draw the number using the text command, at `0,height`:

```
textSize(1.5*width/digits);
text(number, 0, height);
```

For this example, what we will do is initially set the number to 0. Then, we increase it by `mouseX` at the end of the draw loop. This way, the number keeps getting larger.

The hard part here is, given a number, how do we know how many digits are in it? One way to do this is to convert the number to a string, and then get the length of the string. However, we can do it purely numerically, but it requires a bit of trickery. Notice that for a number *n*, if we divide by 10 using integer division, we effectively remove the right-most digit:

```
number = 1234;
number /= 10; // now has 123
number /= 10; // now has 12.
```

If we keep doing this, what happens? Eventually, the number reaches zero. Each time we divide, one digit comes off. If we count how many times we had to do the division before hitting zero, this tell us how many digits the original number had.

So, given a number, let's repeat the above operation while it is not zero, and each time we strip a digit, let's count it.

```
// the variable number has a valid integer
int digits = 0;
while (number != 0)
{
  number /= 10; // strip the right-most digit.
  digit++; // count the stripped digit
}
```

We need a while loop here because we don't know how many digits are in the number, so we don't know how many times the loop will run. We cannot use a `for` loop. If we knew how many digits there were, we wouldn't need the loop at all!

There are a couple of gotchas here. One, our `number` variable should be getting larger each time it draws, by adding `mouseX` to the value (see the previous page). However, our operation of counting digits destroys the `number` value by continuously dividing off variables. Our digit-counting operation is destructive, and so we lose the original number.

The way to fix this is to copy number, and then destroy that copy to count the digits. The original number stays unchanged.

Here is my draw loop, note that number is a global variable that starts at 0.

```
void draw()
{
  background(0);

  int copy = number;
  int digits = 0;
  while (copy != 0)
  {
    copy /= 10;
    digits++;
  }

  textSize(1.5*width/digits);
  text(number, 0, height);
  number+=mouseX;
}
```

Try running the program now.

Oh no! This crashes and doesn't work. What is going on? The algorithm above actually works pretty robustly, except in one specific case! We have a special case here. What happens when `number` is 0? We expect the result to be 1 digit. However, in this case, since `number` starts at 0, the while loop is never run! Since the copy variable is 0, then `copy != 0` is false, and the loop is skipped. No digits are counted. How can we fix this?

The easiest way is to use an `if` statement to keep track of the special case. If number is 0, then we have 1 digit. Otherwise, use the algorithm to calculate it.

As you will often see, algorithms have dusty corners where they do not work, and often require special cases. You can usually fix these by addressing them individually. However, if you have many special cases, then perhaps you can find a better way to solve the problem.

Here is my code:

```
int number = 0;

void setup()
{
  size(500, 500);
}

void draw()
{
  background(0);

  int copy = number;
  int digits = 0;
  if (number == 0)
  {
    digits = 1;
  } else {
    while (copy > 0)
    {
      copy /= 10;
      digits++;
    }
  }

  textSize(1.5*width/digits);
  text(number, 0, height);
  number+=mouseX;
}
```

12.9 **The “do-while” Loop**

Sometimes, the initial condition setup for a loop can be quite extensive, and essentially becomes all the code inside the while loop. In this case, sometimes you’d just like to ask Processing to run the whole loop one time first before checking the condition. With the while loop, if the condition is false, the loop is never run, not even once.

There is another kind of loop that helps mitigate this issue called the “do-while” loop. This loop is not used heavily in practice. However, you should know about it as you may come across it in other people's code.

```
do
{
   // statement;
   // priming / upkeep
} while (condition);
```

This loop always runs at least once. In a while loop, if the condition is false, it is never run. Since the loop is run at least once you can put the priming and upkeep as one block just before the test.

Do while loops have some issues. What if the loop never should be run, not even once, e.g., if the condition is false from the get go? Too bad, it runs at least once. Also, since they are used less often, people can find them confusing, which can lead to bugs.

12.10 Loops and Scope

Consider the following code:

```
for (int i = 0; i < 999; i++)
{
      do some stuff;
}
println(i);
```

What will the output be? Well, `i` starts at `0`, and gets larger, and the loop runs while `i<999`. So, if `i=998`, then it's still true... if `i=999`, it becomes false, and the loop stops. So, the output should be `999`, right?

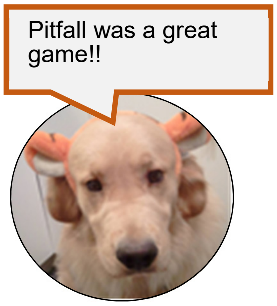
Kind of. That would be the output if this code actually ran. Remember scoping rules? A variable that is created inside a block only exists inside of that block. When you declare a variable inside the for-loop initializer, as in this case, it becomes part of the for loop's local scope. That is, once the `for` loop has ended, the variable is destroyed and cannot be accessed anymore.

The same goes for the while loop. While you do not create a variable inside the while loop condition, any variables you make inside the block get destroyed at the end of that block.

What about the following example? We know that we cannot have more than one variable with a given name…

```
for (int i=0; i < 200; i++) {
      // some code!
}
for (int i=0; i < 300; i++) {
      // some code!
}
```

Does this work, or, does Processing complain because we have two variables with the same name? In fact, this code is perfectly fine. Since the `int i` variables only exist within their local scope, they do not interfere with each other. Processing forgets about the old `int i` the instant the loop is over, so when we try to create a new one, it's not a problem.



12.11 Common Loop Pitfalls

There are some common mistakes made with loops, and it's worth peeking at them here.

Sometimes, the code inside a loop is never run, not even once, and it can be hard to see why. For example:

```
for (int i=0; i>= 10; i++)
{
// do something
}
```

A common cause for this is that loop test is never actually true. If the test is not true, the block is not run, and the loop is ended before it starts! In this case, since `i` starts at 0, then the test `i>=10`, which resolves to `0>=10`, is clearly false, and the loop is ended. Make sure that your starting condition is conducive to actually running the loop.

Similarly:

```
int i = 10;
while (i > 100)
{
      println(i);
      i--;
}
```

In this case, the while condition `i > 100` is never true because `i` starts at 10.

Another problem is that a loop can sometimes keep running seemingly forever even though it should be short:

```
for (int i=0; i<= 10; i--)
{
  // do something
}
```

In this case, the problem is that `i` is getting smaller, not larger, so the test `i<=10` doesn't get closer to being true. Does this run forever? Think about the limits of memory, and overflow, discussed in Unit 9. Make sure that your variable changes appropriately and moves toward your end condition.

There are a lot of ways to mess up so that you never get a false check condition. Consider the following:

```
int i = 1;
int j = 1;
while (i < 1000)
{
      println(i);
      println(j)
      j++;
}
```

This doesn't work because j keeps getting larger, but `i` stays at 1. The test of "is `i<1000`" is always true! The loop never ends.

How would you fix this? Either change the while condition to check j, or change the loop upkeep to be sure that `i` gets larger.

What about this example?

```
int numTimes = 0;
while (numTimes <= 10)
{
      println(numTimes);
}
```

What do you expect to happen? This will be an infinite loop because the loop is missing an upkeep. `numTimes` stays at 0 and the test is always true, so it will print out the message until the end of time.

Sometimes, your loop simply won’t compile:

```
boolean onScreen;
while (onScreen)
{
  …
  onScreen = (ballX >= 0 && ballX < width) &&
     (ballY >= 0 && ballY < height);
}
```

The `onScreen` variable is created at the loop priming spot, and then it is checked right away. Is the value `true` or `false`? Does the loop start or not? We have no idea. No data has been placed into the variable, so the while loop cannot even check it. This doesn’t work. Similarly:

```
for (i=0; i<= 10; i++)
{
// do something
}
```

While everything here looks reasonable, the variable `i` was never declared, and as such Processing doesn’t know what to do with it. Don’t forget to declare your variables!

Important: There is one more pitfall with loops. Be careful when using floating point variables. In fact, I will go as far as saying that ***you should never use a floating point as a counter in a loop,*** particularly for loops.

Consider the following example:

```
for (float f = 0; f<=1; f+=0.1)
{
  println(f);
}
```

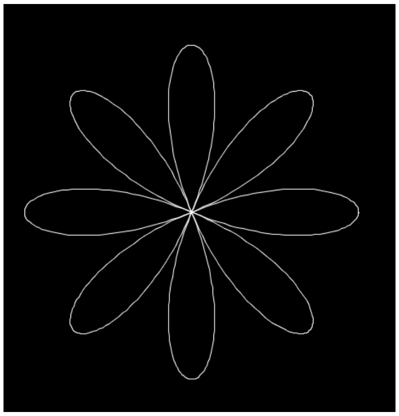
What do you expect to happen? If you did this on paper, you would get 11 lines output: `0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0`. However, since floating point numbers cannot be trusted to be accurate, we get rounding error. This loop only outputs `10` lines, not `11`, and `1.0` is never output. Try it. Never use floating point variables in a `for` loop.

How would you fix this? Use an integer to iterate, and then a simple calculation to get what you want. This way, any error does not accumulate, and you get an exact count of iterations.

```
for (int i = 0; i < 11; i++)
{
  println(i*0.1);
}
```

Try it out. As you can see, you still get precision error. However, it does not accumulate. Further, you get exactly 11 lines of output as you expected.

12.12 **Example: Rose Art**



In Unit 7, exercise 5, you generated an animation of drawing rose art. At that time, doing the art incrementally was the only way to get the job done. Now, with for loops, you can draw the whole rose at once. As a reminder, here is the formula to generate a rose. *k* determines the number of petals, and x_c and y_c determine the center of the rose.

$$\mathrm{x} = \cos(kt)\cos(t) * r + x_c$$

$$\mathrm{y} = \cos(kt)\sin(t) * r + y_c$$

This is a parameterized function on *t*, and we draw the rose as `t` goes from 0 to `2 PI`. At `t` of 0 we get the `x,y` of the starting point, which should be the same as the final point, at `t = 2 PI`.

To draw the whole rose, we need to generate points along the rose at different values of *t*, and draw lines between them. For the most part you should be able to try this out yourself. The tricky part is how to use a loop to help out with this.

There are several ways to approach this. One way is to use a floating point variable, start it at 0, and choose some increment to make `t` go from 0 to `2PI`, for example:

```
for (float t = 0; t <= 2*PI; t += 0.01)
```

And this will generate a range of t values. However, as we said above, you should avoid using floating point variables in a `for` loop due to accumulation of precision error. Instead, re-think the problem in terms of integers.

The way that you approach this is to determine how many points along the rose you want to calculate. Let's call this `STEPS`. If you want 100 points, then we use a for loop to go through those steps. This avoids the floating point issues:

```
for (int i = 0; i < STEPS; i++)
```

Then, you use the value of i to calculate a t somewhere along the range of 0…2PI. This is called linear scaling, and the simplest way to do it is to a) convert i into a percentage along your range, and the b) take that percentage of your target range.

```
for (int i = 0; i < STEPS; i++)
{
  float percent = i / (STEPS-1);
  float t = percent * 2*PI;
}
```

Why do we add -1 to STEPS? Think that one through. If you use it or not, changes the distribution, and the final value of *t* in the last loop.

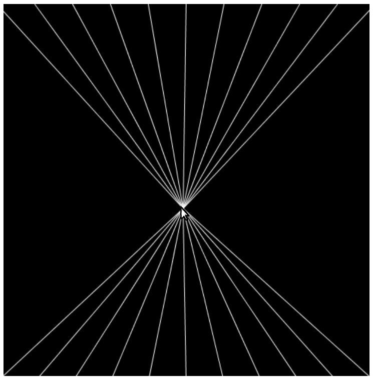
You can finish this example on your own.

Check your Understanding

12.13 Check Your Understanding: Exercises

Exercise 1. Make for loops to output the following series of numbers to the console, using println. All ranges are inclusive of the start and end number Make sure to run them to test them, as you will often have off-by-one errors. The only command inside the for loop should be the println, with everything else happening in either the initial condition, upkeep, or test. Hint: these often show up on tests.

a. 1..20
b. 25..30
c. 10..0
d. 1..100, even numbers only
e. -50..50

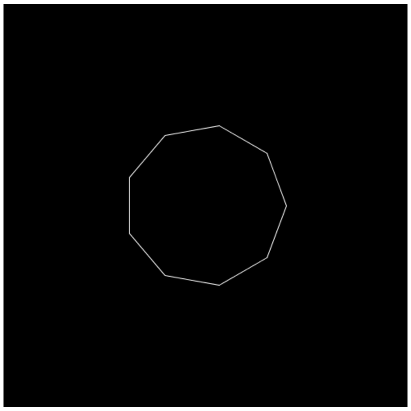


Exercise 2. Make a program, like in the inset, which draws lines from the top and bottom edges of the screen, to the mouse. Space them 50 pixels apart. Make sure to use a for loop to save work, don't draw each line command individually.

Exercise 3. Update the clock example from 12.4.

a. add minute and hour hands. You will have some confusion to make sure that, e.g., 12:00 is straight up. Remember that 0 degrees is right along the positive x axis

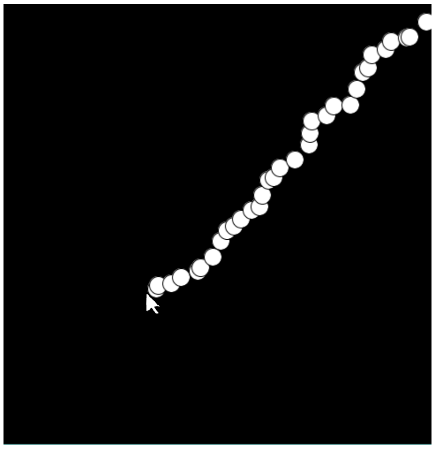
b. make the second hand move smoothly between the tick marks using the `millis()` command, which gives the number of milliseconds from the clock. Warning: this is the number if milliseconds since the program started, not within the current minute or second, so you have some tricky math to solve.



Exercise 4. Computers do not draw perfect circles. Because there is only a limited number of pixels, a circle in a computer can only be an approximation of a perfect circle. Usually, any smart program does a good job and uses a lot of tricks to make it look good enough that you see it as a perfect circle. At the right, you see a circle approximated by 9 line segments (a 9-sided polygon). As more sides are added, the circle gets smoother. In this exercise you will draw polygons of various sizes, controlled by the mouse, to approximate a circle. Use a radius of 100 (diameter 200) as shown.

a. First, use a `for` loop to draw the example given here, with exactly 9 points. The starting point for the first is at angle 0 (straight out to the right). Every point after that will be at an angle of, in this example, 1/9th, 2/9ths, 9/9ths of the way around the whole circle (2π radians). Use a `for` loop to go through the point numbers (use integers!) then calculate the radians from that number. Use a line to connect each point to the previous point. You will have to keep track of the previous point using variables. Remember that for a circle with center (xc, yc), radius r, the point at an angle of θ radians around the circle is $(\,x_c + r\,cos(\theta),\ y_c + r\,sin(\theta))$.

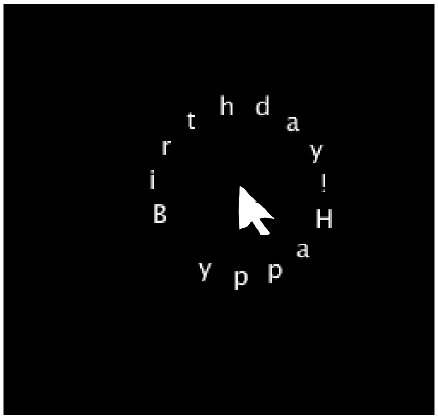
b. Once a is working, use the value `mouseX/20` to determine the value of points instead. In the `draw()` block, erase the window and draw a new polygon every time. Watch what happens as you move the mouse from left to right.



Exercise 5. Make a random walk variant (similar to example 12.6) to simulate a string floating up from the mouse. In each frame, place a circle at the mouse of size 20, then use a `while` loop to place a number of circles above and to the right of the mouse. Do this by moving the ball right and up a random amount from 0 to the radius, and drawing a ball at the new location, while the ball is still on the screen. When this works, it will move quite nicely looking like it is fluttering in the wind.

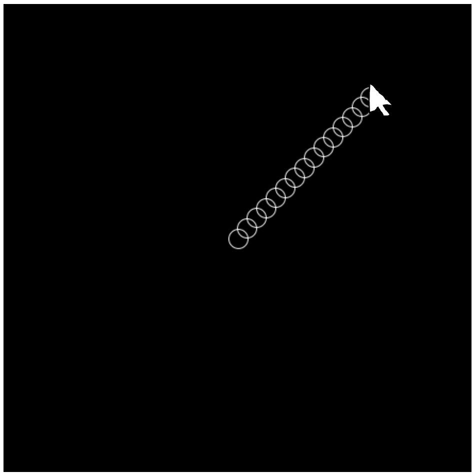
Exercise 6. Update exercise 7 from unit 8 (the random dot example), and use a `for` loop to generate 20 random dots.

Exercise 7. In this exercise, you will draw a message in a ring around the current mouse position. The message will rotate slowly clockwise, but if the mouse button is pressed it will rotate counterclockwise instead. Use a 500x500 canvas, with white text on a black background. The characters should be 15 pixels in size, and they should be evenly spaced around a circle 50 pixels in radius. Use the message "Happy Birthday!", but your program should work with any other String of characters, of any reasonable length, with no other modification.



a. First, use a `for` loop to draw the message, one character at a time, with the characters evenly spaced in a ring around the mouse location. Draw the first character at an angle of 0 (directly to the right). Make sure this works before continuing. Hint: use the length of the string, and math similar to Exercise 4.
b. Second, cause the ring of letters to rotate, by changing the starting angle of the ring by 0.02 radians per frame. If the mouse button is not pressed, make the ring of characters rotate clockwise, but change this to counterclockwise if the mouse button is pressed.

Exercise 8. Write a program that will draw a "chain" of circles that stretches from the center of the screen to the mouse position. Use a global constant to control the diameter of these circles (try 20, which was used in the example). Calculate the appropriate number of circles to use so that the line of circles will always have the appearance of a chain. Use only a few circles if the mouse is close to the center of the window, and more circles if it's near the edge. Try to keep the distance between the centers of adjacent circles as close as possible to 75% of the diameter of the circles. (Make a global constant for this "perfect" distance, too.) The first circle should be exactly at the center of the window, and the last one exactly on the mouse. To get perfect spacing you might need something like 9.26 circles, which of course you can't do. Use the built-in round(x) function which will round a float value to the closest integer. That will work better than simply throwing away the fraction.

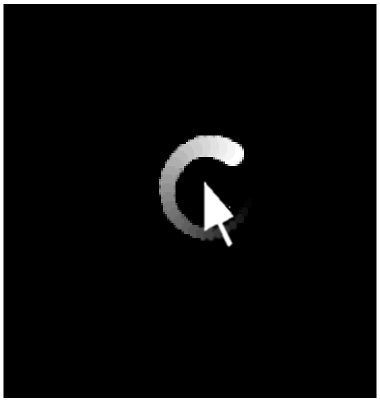


a. Use a `for` loop to draw the appropriate number of equally-spaced circles from the window's center to the mouse position. Do this in the `draw`() block and erase the window every time.

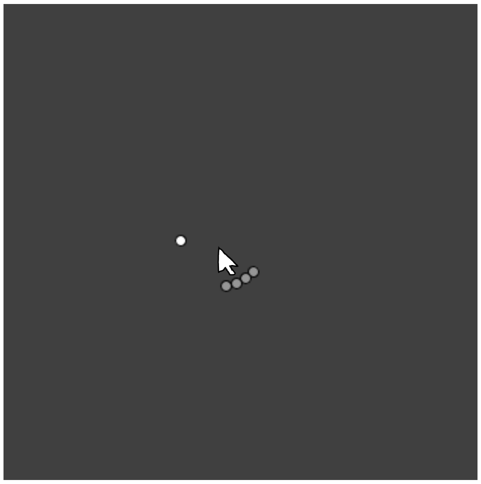
b. Instead of using a fill color, you can tell Processing to not fill the circle at all (make the circle see through) using the command `noFill();` . Now it really looks like a chain.

You don't need to use any trigonometry in this question (although you do need to find the distance from the center of the window to the mouse). The x values for the centers of the circles will be equally spaced, and so will the y values. Find the correct spacing to use in each direction with simple arithmetic, and you won't need anything else. Watch out for off-by-one errors!

Exercise 9. A lot of fancy cursors are used these days, with cool animations, to keep you from getting impatient while you wait for your computer to do something. (I'm not sure it really works.) Here's one for you to try to implement.



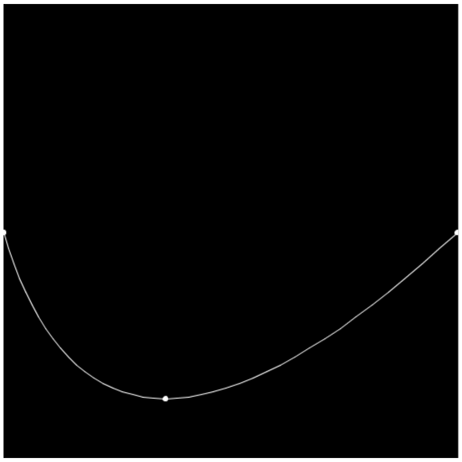
a. Make a ball that circles the mouse. We have seen this before. You do not need a for loop for this.
b. Use a for loop to draw *n* balls behind (in a circle shape) that ball. Just calculate the changing angle at some delta and draw the new ball at that location
c. Change the ball size as you go along the for loop to make it shrink
d. Change the ball color as you go along the for loop to make it fade out



Exercise 10. Here is another mouse cursor to implement. A static picture is shown on the right. There are N small circles (dots) in a ring around the mouse cursor. All of them are moving slowly clockwise. N-1 of them are in a tight group, touching each other, and moving at the same speed. One of them (the white one) is not with the others, and is moving twice as fast. When this "rogue" dot catches up to the group of dots, then it should join onto the trailing end of the group. The leading dot should now break away and become the "rogue" dot, moving twice as fast. This will repeat forever. The effect is like "Newton's cradle" (look it up in Wikipedia), except in a circle. This might keep you entertained while you wait for that huge file to download.

a. Use constants to define the number of dots (2 or more), the diameter of the dots, the radius of their orbit around the mouse, their speed (change in angle per frame), and the angle between the dots in the connected group. You can experiment to get good values. The posted example uses `5, 8, 32, PI/60`, and `PI/12`, respectively.
b. You might want to use a multiplier other than 2 for the rogue dot's speed, relative to the others, but this is not required.
c. First try to get N-1 dots moving correctly around the mouse, without the "rogue" one. Your program should handle any value of N (≥2) without changing any other code at all. This is important.
d. Then add one "rogue" dot moving twice as fast (but just passing right through the others).
e. Then try to detect when the "rogue" dot hits the others, and implement the required effect. That's the tricky bit, but it shouldn't take much code. Good luck!

Exercise 11. Drawing nice rounded curves on computer screens is actually quite challenging, and there are a lot of problems and a lot of ways to do it. You will implement a parametric quadratic curve – a curve that uses a quadratic function to try and fit three different points on the screen to a curve. This is quite a challenging exercise.



The way that parametric curves work is to define a variable, called *t*, that specifies how far along the curve you are (e.g., as a percentage), and then to use some formula to calculate an *x* and *y* for that *t*. As you move along the curve (as the *t* goes from 0...1), you get `x,y` points. If you draw lines between these points, you get an approximated curve.

How smooth the curve is depends on how many points you draw. This, in turn, depends on how small your steps of *t* are. For example, you could imagine large steps of *t* only giving you four points: t=0, t=.25, t=.75, t=1. This is not a smooth curve.

First, we need to define what are called control points of the curve. These are the three dots in the image shown, and they define points along the curve. We will call these `pt0` (start point), `pt1` (some mid-point), and `pt2` (end point). In your program, you will have two variables for each point: one for the *x* component, and one for *y*. In my example, `pt0` is `(0,height/2)`, `pt1` is `(mouseX,mouseY)`, and `pt2` is `(width-1,height/2)`. Make sure to update `pt1` each draw as the mouse moves!

Once you have your control points, we need to first solve for a quadratic function (a parabola) that fits all three points. As you have hopefully learned, any two points define a unique line, and any three points define a unique parabola. Here is a formula for solving this. How that works (and how to parameterize a function) is beyond this course – take more math! ☺.

We need to first calculate the parameters of the curve. These are three points, `a0`, `a1`, and `a2`, which will be the coefficients of our quadratic formula. These rely on a `t1` constant. Set `t1` to 0.5 for now as a floating point variable constant. (once you get the whole curve working then try `t1` of 0.25 or 0.75 and you will see what this value does!). You now have enough information to solve for your coefficients using the following equations.

$$a_0 = pt_0$$

$$a_2 = \frac{(pt_1 - pt_0) - (pt_2 - pt_0)t_1}{t_1 * (t_1 - 1)}$$

$$a_1 = pt_2 - pt_0 - a_2$$

This can be a little challenging. The first confusing point is that we don't have a `pt1` or `pt2` variable, etc., since these both have *x* and *y* components. To do the above calculations, you need to do it twice, one for the *x*, and one for the *y*. This results in *x* and *y* for each of the *a* points.

Now that you have the coefficients, here is your quadratic formula:

$$outPoint = a_2t^2 + a_1t + a_0$$

Whew! This is a lot of messy math. But, now we have a parameterized curve (a curve in terms of *t*) that defines our output. If we put *t*=0 in the above formula, we get our first point. If *t*=1 we get our last point. For all the values of *t* in between, we get spots along the curve. To draw the curve, we need to setup a `for` loop. Here is my pseudocode:

```
♦ set number of steps along the curve, e.g., to 10.
♦ for 0 <= i < steps:
    ➢ t = i / steps; // percentage along the curve. careful of integer division!
    ➢ Calculate outPointX and outPointY as per above, using this t
    ➢ Draw a line from the previous point to outpoint
```

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 13. ADVANCED LOOPS

Summary

You will see in this unit how loops scale up to solve more complex problems. In this section, you will…

- Work through a range of problems that require loops
- See how loops can nest inside other loops

Learning Objectives

After finishing this unit, you will be able to …

- Use loops to solve a wide range of problems
- Create a nested loop to iterate two variables over a range of numbers

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

13.1 **Introduction**

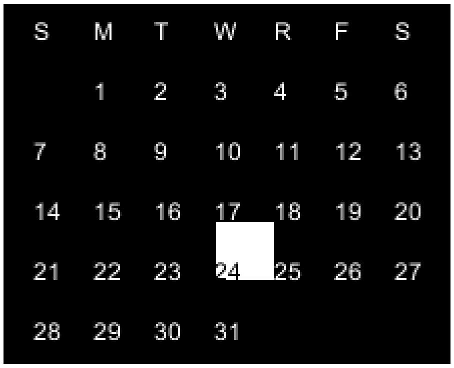
Loops seem to be one of the primary challenges faced by new programmers. The previous chapter introduced you to basic loops and, for the most part, you have seen all the syntax and mechanics that you need. However, where students often struggle is in the application of loops. Even though you may understand the mechanics, it is not obvious to see how to use loops to solve problems, or how loops scale up. For example, what happens if you place a loop inside a loop?

This unit is primarily practice and expansion of the previous unit. However, this doesn't mean that you can gloss over it. Working through for loop examples is a great way to bring all of what you learned together, and to practice your skills on more advanced problems. This is why these two units are so long, because it is very important.

First, let's start with a larger example, and then get into new techniques.

13.2 **Example: Clickable Calendar**

We have now learned enough tools to make a simple clickable calendar. As always, a great place to start is by defining a bunch of constants that help define your calendar, at the top of your program. In this case, we need to determine where the calendar is located on the screen, how many days are in the calendar, and the spacing between the days (the grid). These are the numbers that I used:



```
S    M    T    W    R    F    S
     1    2    3    4    5    6
7    8    9    10   11   12   13
14   15   16   17   18   19   20
21   22   23   24   25   26   27
28   29   30   31
```

```
final int CAL_TOP = 100;
final int CAL_LEFT = 100;
final int CAL_DAYS = 31;
final int CAL_SPACE = 30;
```

Next, let's draw the header row – the titles for the days of the week. You can make this look nicer by putting a background rectangle or changing color, but for simplicity we don't do that here. As always again, helper variables make our thinking a lot easier. Let's set a variable for the left and the bottom of the row – remember that graphical text wants the `x,y` for the BOTTOM LEFT of the text. So, we add one spacing to the top of the calendar to get the bottom of the first row.

```
int bottom = CAL_TOP+CAL_SPACE;
int left = CAL_LEFT
text("S", left, bottom);
```

To calculate the position for the next letter, M, we just add the space to the left variable and draw again:

```
left += CAL_SPACE;
text("M", left, bottom);
```

and so on. Draw all the letters this way. Unfortunately this is a little tedious. The good news is that we'll learn techniques later to simplify this kind of operation.

Next, let's draw the calendar numbers using a `for` loop (why not a `while` loop?). Use a single for loop through all the days of the month. Then, for each day, we can calculate which row and column it is on.

```
for (int day = 1; day <= CAL_DAYS; day++)
{
  // calculate row, column
}
```

To calculate the row and column we can just use integer math. This is a little tricky, but try to work through it. If you divide the day number by 7, it tells you which week you are on (with 0 as the first week). If you take the remainder, it tells you which day of the week (column, with 0 as the first one). This is confusing, so look at this table:

day	d/7	d%7		d/7	d%7		d/7	d%7
1	0	1	**11**	1	4	**21**	3	0
2	0	2	**12**	1	5	**22**	3	1
3	0	3	**13**	1	6	**23**	3	2
4	0	4	**14**	2	0	**24**	3	3
5	0	5	**15**	2	1	**25**	3	4
6	0	6	**16**	2	2	**26**	3	5
7	1	0	**17**	2	3	**27**	3	6
8	1	1	**18**	2	4	**28**	4	0
9	1	2	**19**	2	5	**29**	4	1
10	1	3	**20**	2	6	**30**	4	2

It just works out so nicely! So the row is the day / 7, and the column is the remainder. Actually, we need to add 1 to our row since we already have the header on row 0.

```
int col = i%7;
int row = i/7+1;
```

Now we use that information to calculate the left and bottom location to draw our text. I will add a helper variable in here called `top` since it will help out later.

```
left = CAL_LEFT + col*CAL_SPACE;
int top = CAL_TOP + row*CAL_SPACE;
```

```
bottom = top + CAL_SPACE;
text(str(day), left, bottom);
```

Now we have a nice calendar! The next step is to let the user select a day and to highlight the day that is selected. We can highlight it just by drawing a square as in our calendar image.

To do this, we need a global state variable that remembers or knows which square to highlight. For testing, let's just highlight day 5 by default:

```
int selected = 5; // day currently selected
```

Then, in our draw loop, we need to check which day we are drawing. If we are drawing the selected day, then we put a highlighted square around it. In my case, I set the colors, too.

```
if (day == selected)
{
  rect(left,right,CAL_WIDTH,CAL_WIDTH);
}
```

This only draws a square behind the selected day. Be careful of draw order! Now, how do we change the selected day based on where the user clicks? How can we detect if the mouse is clicked on a particular calendar square? Well, we already know the left, bottom, and top of each calendar square. If we calculate the right side, then we have all the information we need to do hit testing.

Further, instead of doing the match to calculate where the mouse is, we can do it inside the `for` loop on each day. On each day, let's test if the mouse hit it, and if so, update our selected day.

```
int right = left+CAL_SPACE;
if (mousePressed &&
  mouseX >= left && mouseX < right &&
  mouseY >= top && mouseY < bottom)
{
   selected = day;
}
```

Now as the user clicks on a day, it gets selected!

There are two more things we should do. First, the calendar should default to not having any day selected. The way we do this is to set our selected initially to an impossible day, so that our `if` statement comparing to the day never gets true. 0 is an impossible day, so change the global definition to

```
int selected = 0; // impossible day, nothing selected
```

The final thing to do is to enable the person to *unselect* the day. How about, if the person clicks but they do not click on the calendar, then we unselect the day? How can we do this?

This is a little tricky. What we can do is this. Before going through drawing the calendar and checking every single day if it was clicked, we create a boolean that tells us if a day has been hit or not. Let's default it to false, as no day was hit.

```
boolean hit = false;
```

Then, as we go through the calendar, if we ever get a hit on a day, we set that boolean to true. Finally, after we check all the days, if we got NO hit, but the mouse is pressed, then we unselect the day. That's it! We now have a fully interactive calendar. A lot of new things popped up in this example, so be sure to study it. Here is my final code:

```
final int CAL_TOP = 100;
final int CAL_LEFT = 100;
final int CAL_DAYS = 31;
final int CAL_SPACE = 30;
int selected = 0;
void setup()
{
  size(500, 500);
}

void draw()
{
  background(0);
  fill(255);
  // draw title bar
  int bottom = CAL_TOP+CAL_SPACE;
  int left = CAL_LEFT;
  text("S", left, bottom);
  left += CAL_SPACE;
  text("M", left, bottom);
  left += CAL_SPACE;
  text("T", left, bottom);
  left += CAL_SPACE;
  text("W", left, bottom);
  left += CAL_SPACE;
  text("R", left, bottom);
  left += CAL_SPACE;
  text("F", left, bottom);
  left += CAL_SPACE;
  text("S", left, bottom);
  left += CAL_SPACE;

  boolean hit = false;
  for (int day = 1; day<= CAL_DAYS; day++)
  {
    int col = day%7;
    int row = day/7+1;

    left = CAL_LEFT+CAL_SPACE*col;
    int right = left+CAL_SPACE;
    int top = CAL_TOP+CAL_SPACE*row;
    bottom = top+CAL_SPACE;

    if (mousePressed &&
      mouseX >= left && mouseX < right &&
      mouseY >= top && mouseY < bottom)
    {
      selected = day;
      hit = true;
    }

    if (selected == day)
    {
      rect(left, top, CAL_SPACE, CAL_SPACE);
      fill(0);
    } else
    {
      fill(255);
    }
    text(str(day), left, bottom);
  }
  if (mousePressed && !hit)
  {
    selected = 0;
  }
}
```

To expand this example, try setting which day of the week the calendar starts on, or fixing the highlighting to be better centered. Additionally, how would you highlight weekend days?

13.3 **Nested Loops**

Loops can be tricky. As I said before, an important element of really getting loops is to memorize the mechanics – what happens at what stage. This is particularly important as we move into nested loops: you can put a loop inside another loop.

In a previous example, we drew a grid of lines using a `for` loop – we could iterate across the gaps, and draw one line at each gap. For a 10x10 grid, this took 22 lines. What if we wanted to draw a grid of small dots? Let's say we want a grid of 10 by 10 dots, which requires 100 dots. How would you go about doing it?

You could do it with one for loop, if you iterate `i` from 1...100, and then determine the grid coordinates depending on the `i`. This is similar to the calendar example above. Perhaps a more straightforward method is to use a nested for loop.

Note: since a code block acts like any other code, you can put a loop or if statement inside of any loop or if statement. Nested just means one thing inside another.

What happens if we put one for loop inside another? There is nothing special here, all the same rules of programming apply. However, it becomes a little tricky to keep in your head. Look at the following code:

```
int count = 0;
for (int i = 0; i < 10; i++)
{
      for (int j = 0; j < 10; j++)
      {
            count++;
      }
}
println(count);
```

Both of these for loops should make sense to you. The first one iterates `i` from 0...9 (remember – the loop quits when `i` <10 is false!). The second loop iterates j from

0...9. But how do they combine? Keep in mind that computers work on the mechanics. After the `i` loop is started, it runs the block. Inside the block, it starts the j loop. ***The catch is to realize that, each time the i-loop block is run, the whole 0...9 j loop runs.*** Let's step through

- count is set to 0
- the `i` for loop initiates, `i` is 0, block is run
  - the `j` loop initiates, `i` is 0, `j` is 0, block is run
    - count gets increased to 1.
  - the `j` loop checks `j<10`, true, `j` becomes 1, block is run
    - count gets increased to 2.
  - … continue until `j<10` is `false`
- the `i` loop checks `i<10`, true, `i` becomes 1, block is run again
  - the `j` loop initiates, `i` is 1, `j` is 0
  - the *entire* `j` loop runs again.
- … repeat while `i<10` is true

As this illustrates, the inside j loop gets completely run for each time through the outside block.

Study the above steps. This is very important. One thing that I recommend for people to do is to try to solve a nested for loop on paper: first, make a chart of the current `i` and `j` values. Then, go through the loops step by step and record all the changes that happen to those variables.

What does print output? What is the final value of count? To figure this out, we do some basic math.

The `i` block runs 10 times. Each time the `i` block runs, the j block runs 10 times. Therefore, the j block runs 100 times total, and increments count 100 times. Count starts at 0, so count ends at 100.

To test this, try running the following code:

```
for (int i = 0; i < 10; i++)
{
  for (int j = 0; j < 10; j++)
  {
    println(i+" "+j);
  }
}
```

Here, the output shows you what happens to `i` and `j`. Is it what you expect? What changes if you reverse the order of the `for` loops?

13.4 **Example: A Dot Grid**

Use a nested loop to draw a grid of ellipses, 10 by 10. First, we make one `for` loop to go through all the columns.

```
for (int i = 0; i < gridSize; i++) // go through the columns
{
}
```

Let’s also make a loop to go through all the rows. Since we will be combining these, let’s use a different variable name

```
for (int j = 0; j < gridSize; j++) // go through the rows
{
}
```

Now we can combine these loops. The way we should think about it is as follows.

```
For each column in our grid…
    Go through each row in the grid…
        Draw an ellipse
```

Turning this into code:

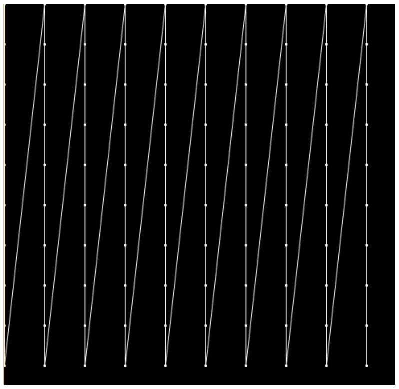
```
for (int i = 0; i < gridSize; i++) // go through the columns
{
   for (int j = 0; j < gridSize; j++) // go through the rows
   {
     // draw the ellipse
   }
}
```

As in our previous example, the whole for loop on j gets run for *each value of* `i`. So, when `i = 0`, we will get j all the way from 0...9. Then, when `i =1`, we will get the whole range on j from 0...9 again. This way, we will hit every combination of `i` and `j` in the 0...9 range.

To draw the ellipse, we just need to convert from the grid coordinates to the screen coordinates. We have done this before. First we calculate how wide each grid cell is, and multiply it by how many grid cells along we are:

```
int x = width/gridSize * i; // column
int y = height/gridSize * j; // row
```

```
ellipse(x,y,2,2);
```



There is one final thing we can try. Even though we have a grid of 100 dots, and, we see the entire grid at once, the computer only draws one dot at a time. We can do a *trace* to see the order that the dots are drawn in. We can do this by drawing a line from the previous point, to the current point. That way, we can follow the line to see what order the points were drawn in. Try implementing this on your own, and then try changing the order of the `for` loops (`i` and `j`) to see what happens.

13.5 **Example: Raster**

"Raster" sounds complicated, but it's just the idea of setting each pixel color on the screen individually. This is how pictures work – it stores a brightness or color for each dot, and draws them on a grid, and you get a picture. Let's play with setting each dot's color based on some calculation to see how it looks.

To do this, we will use a nested for loop to go over every single pixel on the screen. Then, we will draw a point at each x and y coordinate. This is very similar to the previous example, except we can setup our loops based on the size of the window:

```
// go over each x coordinate
for (int x = 0; x < width; x++)
{
  // at each x, go over each y coordinate
  for (int y = 0; y < height; y++)
  {
    stroke(random(255));
    point(x,y);
  }
}
```

To recap – the first loop goes through each possible x coordinate across the screen. At each x coordinate (column), we then run down all the possible y coordinates. The whole y for loop is run at *each x*. This way, every pixel on the screen is touched. Here, we give it a random color, so we get what is called white noise.

If you do this as a static program you will quickly see the result. However, if you do it as an active program, you may notice that you program is very slow – if it's not, you have a fast computer! We are doing a lot of work here, and using some pretty heavy-handed techniques. If you stick with computer graphics, you will learn advanced methods to speed this kind of operation up. For now here are two things you can do to make your program reasonably faster:

- Add the command `noSmooth();` to your setup block. This turns off some extra work the computer does to make your program look nice.
- Shrink your screen size. Half the size (250x250) actually has 1/4 the dots!

So far, this is a pretty boring example which sets every pixel to a random color. However, now that we have this basic setup, we can change things. How about we set the color based on how far it is from the mouse? Remember, the high-school math formula for calculating the distance between two points (in this case, the mouse and a point) is

$$\sqrt{(mouseX - x)^2 + (mouseY - y)^2}$$

We can do this easily. We already know how to subtract numbers. To square them, we can just multiply by itself, using some helper variables just for clarity. We just need one additional command, which actually came up in an exercise in Unit 7:

```
float sqrt(float number); // gives the square root of a number
```

And we can develop the following code:

```
float diffX = mouseX - x;
float diffY = mouseY - y;
float dist = sqrt(diffX*diffX+diffY*diffY);
```

Now that we have the distance, let’s just set the point color to the distance:

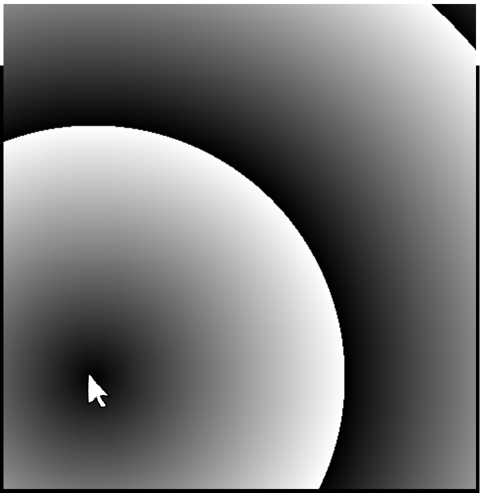
```
stroke(dist);
```

There is one small unsettling thing here, though. The distance on your screen between a point and the mouse may be larger than 255, our maximum color. Processing is very robust and doesn’t mind you doing this, but we should be better behaved and handle it properly. Why don’t we loop the color around, using modulo? That is, before drawing our point, let’s instead calculate the color as follows:

```
float c = dist%256;
```

That is, if `dist` is bigger than 255, c will wrap around from 0 again. Your output should look like the inset on the next page. The hard line from white to black is where the distance equals 256 – this makes a perfect circle, which makes sense, as this defines the same distance from the mouse!

Here is my final program:



```
void setup()
{
 size(500,500);
 noSmooth();
}

void draw()
{
  background(0);
  stroke(255);
   for (int x = 0 ; x < width; x++)
   {
     for (int y = 0; y < height; y++)
     {
        float diffX = mouseX - x;
        float diffY = mouseY - y;
        float dist = sqrt(diffX*diffX+diffY*diffY);
        float c = dist%256;
        stroke(c);
        point(x,y);
     }
   }
}
```

You can play with what formula is used to set the pixel color, which can be a lot of fun. For example, try some of the following ones that I whipped up. Unless you are really mathematical, don't bother trying to figure out why they look the way they do

```
float c = (dist*dist)%256;
float c = (dist*x)%256;
float c = (dist+x-y)%256;
```

```
float c = (dist*x/(y+1))%256
```

Try to animate these visualizations by using a moving offset. That is, add an offset to the colors, and make the offset increase or decrease each frame. This will add a psychedelic effect!

13.6 **Example: Tic-Tac-Toe board**

Let's do a basic tic-tac-toe board. At this point we won't make a game of it, but will just put the basic board in place, and do hit detection with the mouse – to detect which square is being clicked on.

The first part of this example is to draw the board. Since the tic-tac-toe board is only a 3x3, we could probably do it manually. However, it's cleanest if we do it using a nested for loop. The way to think about this is to use two loops – one for each dimension (rows, columns), and then realize that inside the loop, each cell of the game board is hit.

As always, we should make things as general as possible, so let's start by putting some global variables at the top of the program.

```
final int BOARD_GRID = 3; // how many squares
final int BOARD_SIZE = 250; // how many pixels
final int TILE_SIZE = BOARD_SIZE/BOARD_GRID;
int boardCenterX;
int boardCenterY;
int boardLeft;
int boardTop;
```

Why didn't I initialize the center coordinates, or the left and top coordinate? The reason is because these are dependent on the size of the canvas. Since we will change the canvas size in our setup code, we need to wait until the canvas is set before we calculate these:

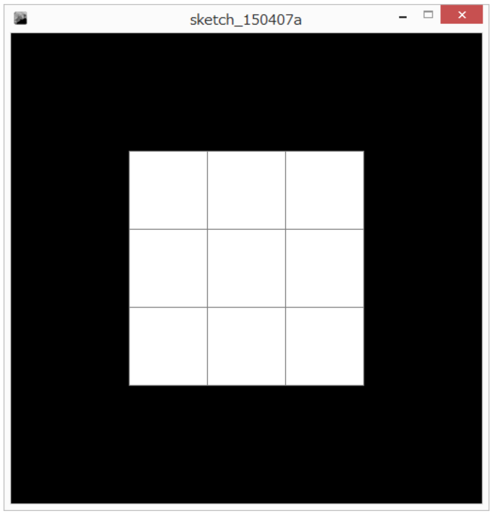
```
void setup()
{
  size(500, 500);

  // center of the screen
  boardCenterX = width/2;
  boardCenterY = height/2;

  // board is centered, so left/top is half size off center
  boardLeft = boardCenterX - BOARD_SIZE/2;
  boardTop = boardCenterY - BOARD_SIZE/2;
}
```

Now, we need to setup our `for` loop to draw the 9 tiles of our board. In this case, it makes most sense to use a nested for loop to hit each tile, with one loop going across the width of the board and the other across the height. First, let's setup our for loops:

```
for (int i = 0; i < BOARD_GRID; i++)
 {
  for (int j = 0; j < BOARD_GRID; j++)
  {
  // draw the tile i,j
 }
}
```

To draw the tiles of the board, we should use the `rect` command. This requires the top and left of the rectangle, as well as the width and height. We already know the width and the height, which we calculated as `TILE_SIZE`. The left and top can be calculated by starting at the left and top of the board, and coming in by how many tiles we are at. That is, the left of tile `i` is `boardLeft + i*TILE_SIZE`. The top is calculated similarly. Inside the nested loop, then we can do the following:

```
int left = boardLeft+ i*TILE_SIZE;
int top =  boardTop+ j*TILE_SIZE;
rect(left,top,TILE_SIZE,TILE_SIZE);
```



I also set the stroke color to 127 with the fill to 255. You should now see the image in the inset.

Now let's work on how to identify which box the mouse is in. Instead of thinking about how to do the math up front, the `for` loop already puts us within each square, one at a time, with the left and top calculated. We can do this right inside the loop and, when the mouse is inside a square, let's change the fill color to 127.

This is just like our previous example of clicking a button, except now we are doing it inside the `for` loop. I would add the following helper variables to make the whole thing clearer. Make sure to add these inside the nested loop since they are specific to the particular square:

```
int right = left+TILE_SIZE;
int bottom = top+TILE_SIZE;
```

Then, the logic to determine if the mouse is inside a square is straight forward. The only gotcha here is the borders. Pay attention to my particular solution and I'll explain below.

```
if (mouseX >= left && mouseX < right && // in X range
  mouseY >= top && mouseY < bottom) // in Y range
{
  fill(127); // a hit
}
else
{
  fill(255);
}
```

The logic is straight forward – if it’s bigger than the `left` AND less than the `right`, we are in the X range. Also, if it’s below the `top` AND above the `bottom`, then we must be inside the tile. In my case, I used the >= on one side and the < on the other side. The equals includes the border, so what I effectively did was to include the left and top border in this square, and exclude the bottom and right border (for the next squares). Quickly add additional logic to test if the mouse is pressed as well. You now have a basic setup for a tic-tac-toe board! Here is my final code:

```
final int BOARD_GRID = 3; // how many squares
final int BOARD_SIZE = 250; // how many pixels
final int TILE_SIZE = BOARD_SIZE/BOARD_GRID;
int boardCenterX;
int boardCenterY;
int boardLeft;
int boardTop;

void setup()
{
  size(500, 500);

  // center of the screen
  boardCenterX = width/2;
  boardCenterY = height/2;

  // board is centered, so left/top is half size off center
  boardLeft = boardCenterX - BOARD_SIZE/2;
  boardTop = boardCenterY - BOARD_SIZE/2;
}

void draw(){
 background(0);
 stroke(127);
 for (int i = 0; i < BOARD_GRID; i++)
 {
   for (int j = 0; j < BOARD_GRID; j++)
   {
     // borders of tile i,j
     int left = boardLeft+ i*TILE_SIZE;
     int top =  boardTop+ j*TILE_SIZE;
     int right = left+TILE_SIZE;
     int bottom = top+TILE_SIZE;

     if (mouseX >= left && mouseX < right &&
        mouseY >= top && mouseY < bottom &&
        mousePressed)
     {
        fill(127);
     }
     else
     {
       fill(255);
     }

     rect(left,top,TILE_SIZE,TILE_SIZE);
   }
 }
}
```

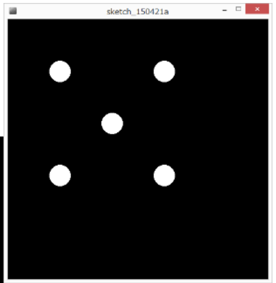
There are a few things to note here. If you hold the mouse button down and drag it across the board, the tile that is highlighted changes and follows the mouse, where you would want only the tile that was actually clicked to change. You can achieve this with some clever logic that you would have seen earlier, in the boolean logic chapters. Another way is to use advanced event handling (look up the `mousePressed()` method function on the reference manual) that we will not learn quite yet.

Also, you may want to save some game state into variables. You would need a variable for each square, and although this is possible with the tools you learned so far, it would be very tedious.

Try updating the board to draw an X or an O in the square when clicked!

13.7 **Example: draw a dice face**

While this may not be the simplest way to do it, this technique for drawing a dice face is great practice for your nested for loops and boolean logic. First, let’s setup a nested for loop that draws a 3x3 grid of dots. Here are my supporting globals:



```
int DICE_GRID = 3; // 3x3 dice face
int DICE_SPACING = 100; // space between dots
int DICE_LEFT = 100; // dice position
int DICE_TOP = 100;
int DOT_SIZE = 40;
```

Then, the for loop setup is straight forward from examples we did previously:

```
// iterate through the columns
for (int i = 0; i < DICE_GRID; i++)
{
  for (int j = 0; j < DICE_GRID; j++) // rows
  {
    // calculate position of dot
    int x = DICE_LEFT+i*DICE_SPACING;
    int y = DICE_TOP+j*DICE_SPACING;

    // draw dot
    ellipse(x, y, DICE_TOP, DICE_TOP);
  }
}
```

This will give you a 3 x 3 grid of dots. To make a dice shape, we need to put an `if` statement in front of the `ellipse`, to only draw dots when certain conditions are met. Let’s think about the dice #3, first, which consists only of a diagonal. Well, how do we define a diagonal in our grid? One diagonal is when `i = j`: `0,0 1,1` and `2,2`. So let’s try changing the ellipse code as follows:

```
if (i == j)
{
  ellipse(x,y,dotSize,dotSize);
}
```

You now should have a 3 dice! Let’s do 5 next, which is a little trickier. First, we can see that we can make a 5 by doing two diagonals. We have the one diagonal, but how can we get the other? Try writing the coordinates out:

2,0 1,1 0,2. Can you see a pattern? This is similar to the diagonal one above except the x coordinate is backwards. As i goes up, 0,1,2, how can we convert that i into 2,1,0? You can take the maximum, 2, and subtract i from it. This is confusing so check out the following table

i	2-i	j
0	2	2
1	1	1
2	0	0

Still confused? Draw it out on paper. Notice how I put j down backwards? This illustrates that when 2-i == j, then our i,j gives us the coordinates we want. So, we can get the coordinates for the opposite diagonal when (2-i) == j.

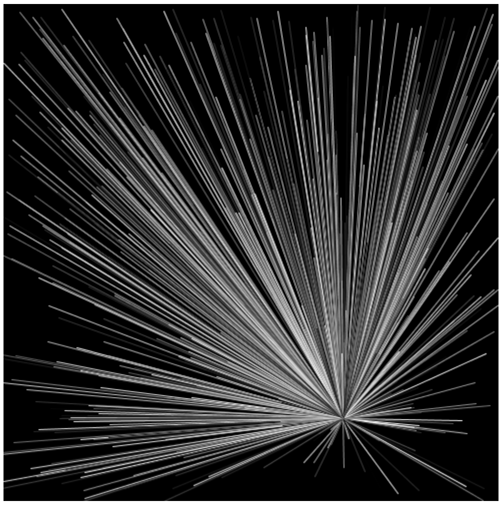
```
if ( (2-i) == j)
{
  ellipse(x,y,DOT_SIZE,DOT_SIZE);
}
```

Finally, we can combine the two diagonals with a simple OR operator:

```
if ( i==j || (2-i) == j)
{
  ellipse(x,y,DOT_SIZE,DOT_SIZE);
}
```

And we're done!

13.8 **Example: Nested While Loop**



While loops can be nested just like other loops, if statements, etc. Let's do a quick example.

We will make a program that acts unpredictably. The interface pauses for a random amount of time, and draws lines to the mouse point – the longer it paused, the more lines you get.

There are a lot of ways to make a program pause. We will use a technique called a **busy loop**, which basically makes work and wastes computer power to kill time. In a later course you will learn why this should be avoided at all costs, but it’s a fun exercise ☺.

In order to wait a random amount of time, we will basically roll dice.

- Pick a random number up to MAX
- If the roll is a 0, we’re done. While it’s NOT zero,
    - Pick another random number.

The larger that MAX is, the less likely we are to get a 0, and the more times the loop runs. Next, let’s add some lines EACH TIME we don’t roll a zero. Our updated algorithm is:

- Pick a random number up to MAX
- If the roll is a 0, we’re done. While it’s NOT zero,
    - Pick another random number.
    - Draw 5 lines from random locations to the mouse.

Keep in mind that while this loop is running, the draw block never finishes, and the screen doesn’t update. So, the longer the while loop runs, the longer the animation freezes.

Implement a while loop for the outer random number guesser first. Approach this with the priming, test, and upkeep mentality. I have ODDS set to 500.

```
int i = (int)(random(ODDS));
while (i != 0)
{
  // draw lines
  i = (int)(random(ODDS));
}
```

We start with a random number. If it’s not 0, we pick another one, and check again. Repeat. This will take some time.

Now we need to update the loop, and each time we get a random number, we need to draw that many lines. The nested loop we could do with a for loop, but use a while for practice. This should be straight forward, so try it on your own.

Check your Understanding

13.9 Check Your Understanding: Exercises

Exercise 1. Re-implement the dot grid example (draw a 2D grid of dots), and make the following changes:

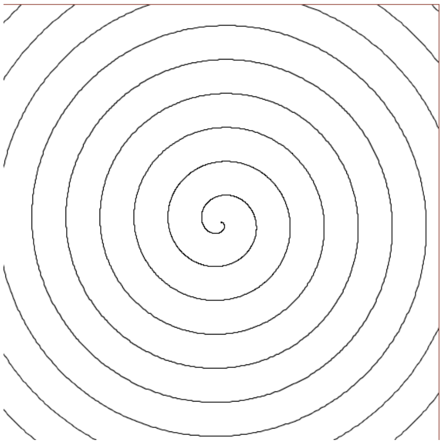
a. Make the size of the ellipse depend on how far it is from the mouse. Figure out the maximum distance, and linearly scale the actual distance to a reasonable size range.
b. Make the color of the ellipse depend on how far it is from the mouse. Use the same logic as in a, but map to colors instead.

Exercise 2. Update the dice face example to include all the usual 6 numbers. In each case, only use one if statement, and try to minimize how many checks you need. 4 may be the hardest, as you can do it with just two equal's tests.

a. Try implementing functions to draw the faces and an if-else chain to turn a number into a graphic. You can now use this in a game.

Exercise 3. Create a nested for loop to iterate over each pixel on the screen, after first clearing the screen to black. Then, use the following logic to choose the color:

a. If x*y is divisible by 2, then make the color white. Otherwise, make the color black. How does it look?
b. Change 2 to 3, then 4, etc., and run the program each time. Do you see a pattern? What numbers give you a grid of perfect squares? Why would this be? (it's tricky).



Exercise 4. Computers might help you hypnotize your friends! You will make a hypnosis-helper tool, as shown in the screenshot at right. In the Exercise 4 in the previous Unit, you drew an approximated circle by using angles that increased in a `for` loop. To make a spiral instead of a circle, increase the radius, too, as you go along. Then the spiral can easily be animated in several different ways. Implement this program in three short phases:

a. Draw a spiral using a single for loop, and basic trigonometry. First, define the following named constants:

`DRAW_STEPS`: The number of line segments that you will draw to make the spiral. Try a value of 250 to start with. This must be an integer variable, so that you can reliably use it in a for loop.

`MAX_RADIUS`: The maximum radius that will be reached when the last line is drawn. Use a 500 by 500 window, and set this value to $250\sqrt{2}$. (That's the distance from the centre to any corner. If you change the window size, you'll have to change this value, too.)

`numTurns`: the number of turns you want in your spiral. This is not a named constant because we will change it later. Try a value of 5 to start with.

Now use a single for loop to draw exactly `NUM_STEPS` lines. The first one should start in the center of the window. Use the command `noSmooth();` in the `setup()` block to speed up this program if it's slow. Each line should connect the previous point to a new point (remember the previous point). The first point is the window's center.

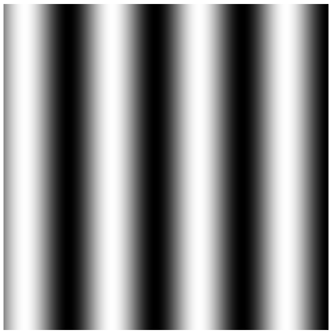
The angle to the new point on the spiral should be 0 for the first line, and increase steadily, by equal steps, to 2π*`numTurns` for the last line. You will need to do linear scaling from your steps to the radians for this.

b. Make the spiral rotate by adding a constant "offset" value to the angle (it no longer goes from `0` to `2π*numTurns`, but instead goes from `offset` to `2π*numTurns+offset`). Start this `offset` at 0, and increase it slightly each time you draw. Try increasing it by 0.1 radians each time (use a named constant, of course). The spiral should now rotate. ("You are getting sleepy.")

c. Also make `numTurns` increase slightly every frame. Try making it change by 0.1 turns per frame (use another global named constant). More and more "loops" of the spiral should be drawn as this value increases. They will lose their smoothness, since more turns are being drawn, but the number of lines used to draw them hasn't changed. This produces a cool effect. Now change your program so that the turns of the spiral stay equally smooth as the number of turns grows. (This isn't as cool, though, so save the old code, too.)

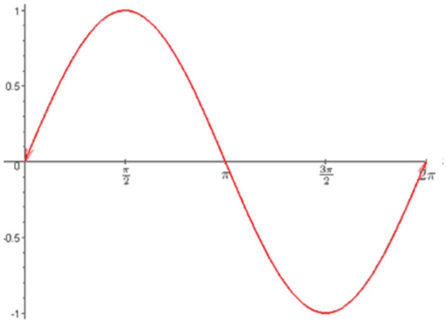
Exercise 5. Use a nested for loop to make a prime checker. Make a global constant called `MAX_CHECK`, and set it to 1000 – this determines how many numbers to check.

a. Make a loop to go from 1 to `MAX_CHECK`, inclusive
b. For each of these numbers, use a second loop to count how many factors it has. Hint: a number `n` is a factor of `p` if it divides evenly into p. (use mod). If the factors are exactly 2, then print out a statement telling the user that the number is a prime.
c. If you increase `MAX_CHECK`, this quickly gets very slow. Can you think of an easy way to speed this up?



Exercise 6. You will visualize what the sine function looks like in "1D" and in "2D". The image at right shows the 1D version with just over 4 cycles. Remember that the sine function is periodic (it repeats) with a period of 2π. As the angle goes from 0 to 2π, the sine function

goes from 0 to +1, then to -1, then back to 0. You will map a sine value (-1 to +1) to 0...255 to obtain a greyscale color. -1 should give black, 0 should give medium grey, and 1 should give white. That is what you see in the image above.

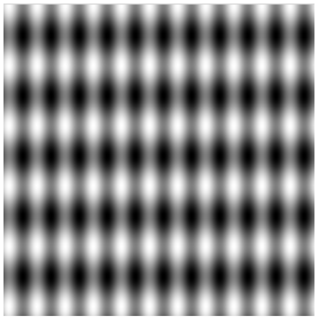


a. Use a for loop and tightly-spaced vertical lines (one per x coordinate across the screen), to draw an image such as the one shown on the previous page. (add `noSmooth();` to `setup()` to increase speed and use a smaller canvas size, such as 300 by 300).

   Set a global variable `numCycles` (in part b it will change, so not a constant) to the number of cycles of the sine function to plot (try 4.0).

   Set the line color based on the x coordinate, by converting the position from `0` to `width-1` to the sine angle, `0 to numCycles*2*PI` (hint: use percentages). Then translate the output from the sine of this value (-1 to +1) to a greyscale value (0 to 255).

b. Now you can move to an animated 1D image. Make `numCycles` controlled by `mouseX` . It should change from `0` to `MAX_CYCLES` as `mouseX` changes from `0` to `width-1`. Try `MAX_CYCLES` at 12.



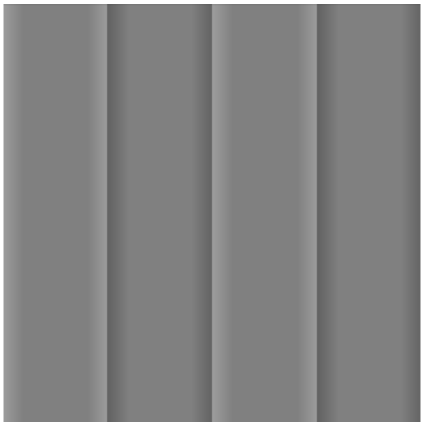
c. Now, we will move to a 2D image. Instead of using lines, you will need to set the color at each pixel using the `point` command. Use two for loops, nested, to hit every pixel. First, simulate question b by setting the color based on the x only, as above. You should get the same output (but probably slower).

   Now, update your program to also do a sine calculation in the y direction. You will need to set the number of cycles in the y direction, and repeat all the sine and scaling calculations to get a y value. First, ignore the x and set the color based on the y only. Once this works, then average the x and y color to get your result. This is what the sine function looks like in 2D!

   Try using color!! Instead of averaging the colors, try using `stroke(xColor, yColor, 0);` which looks cool!

Exercise 7. Optical illusions!! Quick! Are the alternating bands in the image below the same color, or are some lighter or darker than the others? They are actually exactly the same color, with the only color differences being at the borders! This is called the Cornsweet effect. You will implement it so that you know for yourself that the bands are of the same color.

a. Basic image. Use a global constant `BANDS` to control the number of vertical bands in the image. The image at right uses 4 bands. Most of each band will be the same color (128, medium grey), but the bands should either darken or lighten at the edges. Use global constants to control what proportion of each band is the "fade in/out" region on each side, and how much the color will change from the default value of 128. (The image shown uses 20% on each side – 40% total, and fades in/out by 100 – giving grey values from 28 to 228.)



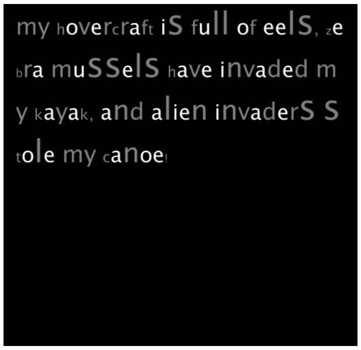
Use for loop(s) to draw the image. You can either use one loop to generate every x coordinate across the canvas, but it may be easier to use two: one for the band number, and one for the position within the band. You can do either one. Use vertical lines, not individual points, to draw the image, spaced tightly so that you draw them 1 pixel apart (or very close to it).

Even numbered bands (if you start at band 0) should brighten from `128` to `128+MAX_CHANGE` on both edges, but odd numbered bands should darken from `128` to `128-MAX_CHANGE` instead. Notice that the change is smooth, with a linear change.

b. Animated image. Link the amount of fade to the `mouseX` value, so that when the mouse is at the left (`0`), there is no fade out at all (so you should get a solid grey screen), and when the mouse is at the right (`width-1`), the bars change completely to white/black at the edges. Now you can see the illusion more clearly.

Exercise 8. You will draw a message, with each character having its own size and color, as shown on the right, with the size proportional to the frequency of the letter in the message. This is not an Active Processing program (there is no animation).

a. Store the message in a global constant, and only use lower-case letters. Determine algorithmically which character appears the most in the text. That is, find the maximum frequency of occurrence of any character in `MESSAGE`. Do not count vowels (a, e, i, o, u) or blanks, since they occur far too often, and spoil the effect. Print the character with the maximum frequency, with its frequency (count), to the console, to test this. In



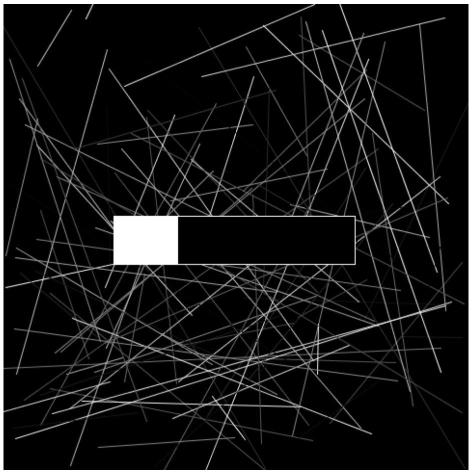
```
my hovercraft is full of eels, ze
bra mussels have invaded m
y kayak, and alien invaders s
tole my canoe!
```

the sample shown, ‘s’ appears most often (7 times), whereas ‘z’, ‘b’, and ‘!’ only appear once.

b. Go through the `MESSAGE`, drawing one character at a time to the canvas. Each character should have its own size and color. Vowels (a, e, i, o, u) should be fixed to `VOWEL_COLOR`, and all other characters should be `TEXT_COLOR`, drawn against a `BG_COLOR` background. A non-vowel, non-blank character having the maximum frequency (determined in part a) should be `LARGEST_TEXT` pixels in size, whereas one that only appears once should be `SMALLEST_TEXT` pixels in size. Others should have a size somewhere in between, proportional to their frequency. All vowels and blanks should have a size halfway between those two extremes, regardless of frequency. To space the characters, you can use the function `textWidth(s)` to find the width `w` of any string `s`, in pixels, at the currently selected font and size. (Because anything can be converted to a String, s can really be any data type at all.) Begin the next character w pixels to the right of the current character, where w is the width of the current character. To test the program so far, draw characters starting at x=0, y=height/2, and let them disappear off the right edge of the canvas.

Phase 3: Draw the message on as many lines as necessary. Draw all lines starting at x=MARGIN. Draw the first line with the characters at y=FIRST_LINE_Y. Whenever you have to start a new line, increase the y coordinate by LINE_SPACING. Draw as many characters as you can on the current line. But if drawing the current character would go beyond width-MARGIN, then start a new line before drawing that character. Now the program should give the result shown in the sample image.

Exercise 9. Make a status bar on a prime number calculator. When computers are working hard, users want feedback on the progress. If the computer waits until it is done all the work before giving an update, the user may be waiting a very long time and may worry that the program crashed. Status bars are a great way of keeping users informed.



You will implement a program that finds the 10000th prime number. Here are my globals: `PRIMES_TARGET = 10000`, `BATCH_SIZE = 100`, `STATUS_BAR_WIDTH = 250`, `STATUS_BAR_HEIGHT = 50`, `primesFound = 1;` // assume that 1 is already counted, `nextPrime = 2`, `done = false`;

We defined a `BATCH_SIZE` – this tells us how many numbers to test if they are prime, before updating the user. Here is a sketch of the program:

- Do a batch (use a for loop) of numbers to test, run the loop `BATCH_SIZE` times
  - Check if `nextPrime` is a prime. See exercise 5 above:
  - Increase `nextPrime` for the next check.
  - if we found enough primes, change our boolean to signal we are done. And don't do any more simulations. NOTE: what if you are in the middle of a batch?? How do you end the for loop early? Hint: modify the for loop condition to also rely on the `done` variable
  - draw a random line with a random color to signify that a number was checked
- After the batch is done, draw the status bar using two rectangles, which shows how many primes were found out of our target maximum. Hint: use two rectangles, one for the whole box, one for the filled amount.

This example illustrates an important usability issue. It is important to give people feedback, but the computer has to do wok to provide this feedback. If you provide too much feedback, it slows the real work (prime checking) down. Play with the batch size to see how the speed changes. Also note that the checking is slower as you go along, because it takes more work to check a large number than a small one.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 14. USER-DEFINED FUNCTIONS PART 2

Summary

You will see in this unit how to use the full power of user defined functions. You will…

- see how to pass information into functions, and information back from calculations done in a function.
- see common pitfalls associated with functions
- learn how to avoid using changing global variables in functions when possible

Learning Objectives

After finishing this unit, you will be able to …

- create a function to take one or more parameters
- create a function that returns data
- have one function call another function
- apply top-down programming to simplify and solve complex problem

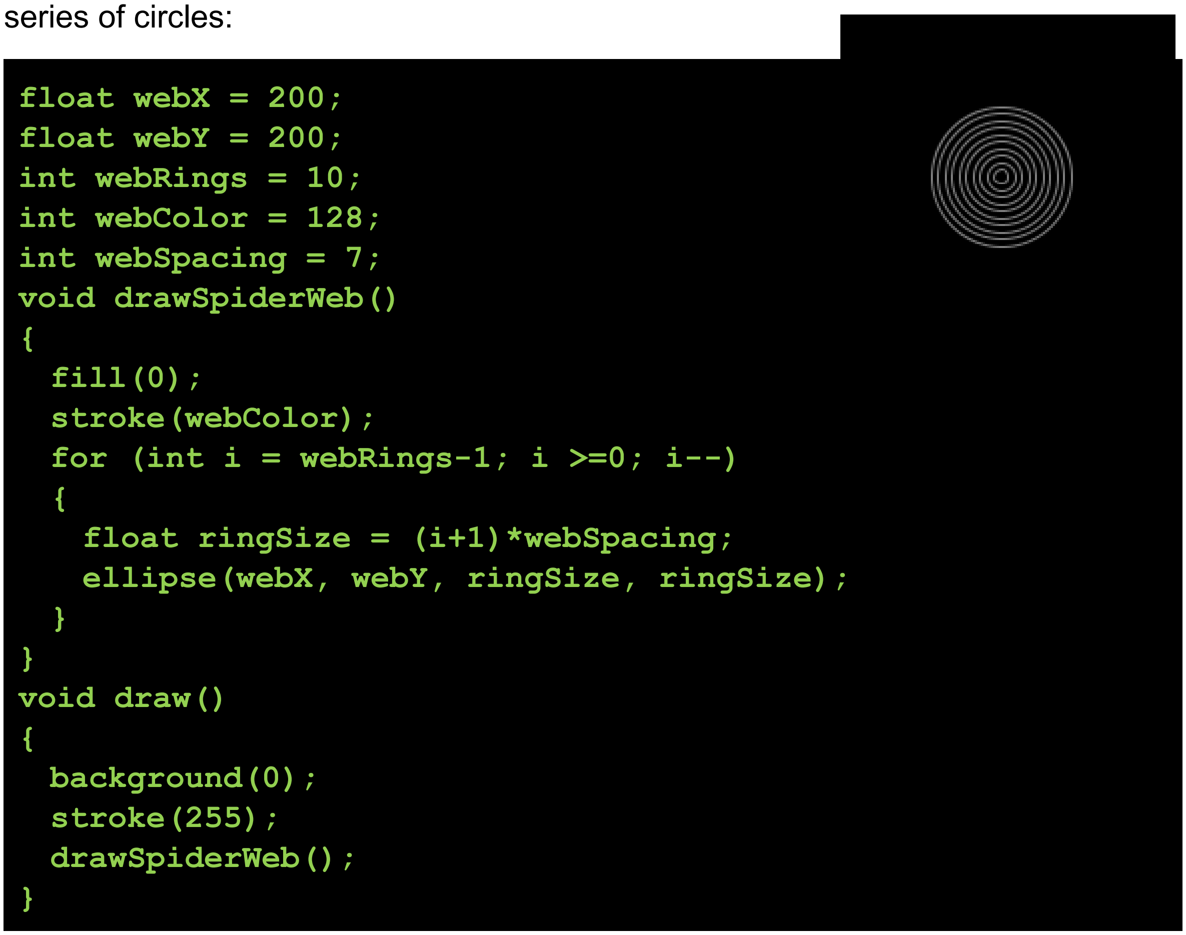
How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

14.1 Introduction

So far, we learned basic user-defined functions as a technique to help us manage our programs as they get larger – we can break a program down into manageable chunks, and work on one chunk at a time. However, these have been very limited. For example, they all rely on global variables to share information, which gets confusing and difficult to keep track of. You also may have noticed that it is hard to make functions that you can easily re-use, as they may require specific data that changes, and you end up writing multiple similar functions.

Here is an example that illustrates the limitation we currently have with functions. Here is a function that draws a spider web, approximated just by drawing a tight series of circles:



```
float webX = 200;
float webY = 200;
int webRings = 10;
int webColor = 128;
int webSpacing = 7;
void drawSpiderWeb()
{
  fill(0);
  stroke(webColor);
  for (int i = webRings-1; i >=0; i--)
  {
    float ringSize = (i+1)*webSpacing;
    ellipse(webX, webY, ringSize, ringSize);
  }
}
void draw()
{
  background(0);
  stroke(255);
  drawSpiderWeb();
}
```

What if we want to draw several spider webs? For example, draw them to the left and the right of the mouse? With the current solution, we need to change all the global variables, and re-call the function, each time. This is not only tedious, it is error prone, as we may forget what we changed the global variables to. There is a better way – we can send information directly to the function, through a special, secret hidden channel!

14.2 Parameters – sending data to functions

It is great to re-use code, but what if we want slight variation on the code? As in the above example, what if we want to do essentially the same thing, but with small changes to some of the variables involved. For example, if we want a spider web at a different location, or with different spacing, but don't want to rely on changing global variables

We need a new technique: we need to add parameters to our function.

Luckily, you have been using commands and parameters for the whole class, so part of this will look familiar. Every time you use the `line` command, for example, you pass it four parameters that the command then uses to create your line. Similarly, we want to be able to pass information to our own functions. For example, instead of calling the following command in our draw block:

```
drawSpiderWeb();
```

We want to send it some information. For the first case, let's just try sending it the x coordinate of where to draw the spider web:

```
drawSpiderWeb(100);
```

If you try to run this now, it won't work, because Processing will complain that the function cannot take any parameters. We need to fix that by modifying the function.

To create a function with parameters, you modify the function header, and add the parameters inside the brackets after the command name. Starting with our simple case of only accepting one parameter, the x coordinate of the web, here is the syntax to make a function take a parameter:

```
void functionName(parameterType parameterName)
{
  //… code
}
```

In this case, we want the function `drawSpiderWeb` to take an integer parameter specifying the x coordinate of the spider web to draw. So, we re-write the function as follows:

```
void drawSpiderWeb(int x)
{
  ///…
}
```

Now comes a tricky part. We have now a) created a function that accepts one integer parameter, called `x`, and b) modified the function call to provide an integer (100 in this case). But, once the data is passed into the function, how do we use the data?

When the function is called, for example:

```
drawSpiderWeb(100);
```

Two things happen. First, inside `drawSpiderWeb`, a new local variable is created called `x`, and second, that variable gets set to the parameter value. So, internally we get the following hidden code:

```
int x = 100;
```

Now, you can use `x` as a local variable, just like any other.

Update your program so that the function uses this new x value, and, your draw block sends in the appropriate data, and make sure it works. You can test it by deleting the global `webX` variable.

Now, instead, replace the `drawSpiderWeb` commands in the `draw` block with the following:

```
  drawSpiderWeb(mouseX+100);
  drawSpiderWeb(mouseX-100);
```

You should see two spider webs always to the left and right of the mouse, although the y coordinate is not yet fixed. As you can see, now we can re-use our code with slight modifications, by sending data to the function. Try changing the code with different values to see it work.

Important: ***Remember the underlying code above- a local variable is created, and the data copied in.*** It will help you see that data is always only copied into a function, and placed into a new local variable. If you modify this value, the original doesn't change, it was only a copy. We will see more of this below.

Further, since this is a local variable, your same scope rules apply:

- The parameter variable is only accessible from within the function
- The parameter variable gets destroyed every single time the function finishes running.

14.3 **Example: status print**

Let's do another example. Let's make a function that can print a status message. It will print some text at the bottom left corner of the screen. In addition it will surround the text with dashes and put the length at the end. For example, if you use

```
statusPrint("System OK!");
```

It will put the text at the bottom left corner of the screen: `-System OK!- 10`

In this case, we make a new function that takes a single string parameter, and then constructs the output text, before printing it to screen.

This should be reasonable to do with the tools you learned so far, so try it on your own. Keep in mind that you need one parameter, and it needs to be of the `String` type. Then, you use that parameter inside the function as a local variable.

```
void statusPrint(String message)
{
  message = "-"+message+"- "+message.length();
  text(message,0,height-1);
}
```

The nice thing about this is that you can use this anywhere in your code to print a status message on the screen. This code is reusable, and, you can send data to it to have your own custom status messages. It does not rely on any changing globals (except the canvas size), and so you can use this in any program.

14.4 **Multiple Parameters**

So far our examples only take one parameter at a time into your function. Luckily, it is very simple to add multiple parameters. All that you do is place a comma between them in the function header:

```
void functionName(type1 param1, type2 param2)
{
  //… code
}
```

You can have as many parameters as you want but most functions only have 1-3. If you find yourself with 10 parameters, there is probably a better way to do what you're trying.

Let's update our spider web example to not only take the `x` coordinate, but also the `y` coordinate. Therefore, we take a second integer parameter for this. Our new function header looks like this:

```
void drawSpiderWeb(int x, int y)
```

And again, inside the function we use the new variable, `y`, like a local variable.

There is one very important point here: the order of the parameters here MUST MATCH the order that you use them when you call the function. That is, if you call `drawSpiderWeb` and give it the `y` first, and then the `x`, it won't work as you expected. It will copy the first integer into `x`, and the second into `y`, to match the setup you have in the function header.

Our updated function:

```
void drawSpiderWeb(int x, int y)
{
  fill(0);
  stroke(webColor);
  for (int i = webRings-1; i >=0; i--)
  {
    float ringSize = (i+1)*webSpacing;
    ellipse(x, y, ringSize, ringSize);
  }
}
```

And in the draw loop, you call it as follows, and the spider web will stick to the left and the right of the mouse.

```
  drawSpiderWeb(mouseX+100, mouseY);
  drawSpiderWeb(mouseX-100, mouseY);
```

You can also mix and match types, just be sure again that the order that you put the parameters in the function call (the command) match how you put them in the function header. Try to update the above example by sending in the spacing of the spider web rings, and the number of rings, as parameters.

14.5 **Getting Data Back from Functions**

Sometimes functions don't only do useful things like draw on the screen, but also can do calculations and checks that provide results that you may want to use elsewhere in your program.

We have already seen a range of function that give you data. `min`, `max`, and `sqrt` are examples of function that not only take parameters, but after the work is done, it sends data back to you that you can use somewhere. Look at the following line:

```
int biggest = max(variable1, variable2);
```

Here, you are sending two pieces of information into the max function as parameters. When max is done, it gives you data back that, in this case, you store the data in your variable `biggest`. Let's see how to provide similar functionality in your own programs, by making functions that can provide data.

First, ***one important point to learn is that functions in Processing (and most languages) can only return one piece of data***. Period. No exceptions. This follows what you have seen in practice: `random` gives one number, `max` gives one number, `min` gives one number, etc.

Advanced: As you gain experience, you will soon see that this limitation is not acceptable. Different languages have different ways of getting around this. Some languages simply allow you to return multiple pieces of data in the language itself. Others, like C, provide a structure called a "struct" that enables you to group data together. However, throughout your CS career, you'll most likely use object oriented programming to overcome this limitation. We use none of these techniques in this course.

Let's start with a toy example. Let's make a function to calculate and return the maximum of two numbers, called `myMax` that we can use like this:

```
int bigger = myMax(10,20); // expect 20 as the result
```

First do this using the tools we already have. Make a function to calculate the larger of two numbers using an `if` statement

```
void myMax(int a, int b)
{
      int max = a;
      if (b>a)
      {
            max = b;
      }
}
```

This function takes two integer parameters, `a`, and `b`, and uses a simple technique to calculate which is larger. It assumes that `a` is larger, but then checks if `b` is, and if so, saves that one instead into `max`.

If you try to run this code as-is, it won't work: when you try to store the result from `myMax` into a variable, as in the upper line, Processing will complain because `myMax` doesn't provide any data that can be stored.

We need to learn two new things to make this function work the way we want it to, to return data:

- Modify the function header to tell processing that the function returns data.
- In the function, specify which data should be returned.

The first point is easy. In the header, currently we have:

```
void functionName(type parameter1, type parameter2, …)
```

The `void` actually means that the function does not return anything. To specify a return, all that we have to do is to change `void` to the return type. In this case, that would be `int`, since our maximum of the two parameters is an integer:

```
int myMax(int a, int b)
```

Now processing expects the function to return an integer when it is done. At this point, if you try to run your program, it won't work. Processing will complain because, although you told it that your function returns an integer, you didn't specify which integer should be returned.

We can specify this with a new keyword, `return`. In the function, we do

```
return someData;
```

The **return** ***keyword does two things: it ends the function call and returns to the part of the program that called it, and it copies the data back to the caller.***

At this point, it is highly recommended that you put the `return` command at the end of the function as the last command. In fact, some instructors deduct marks if you do not do it this way. That is, do not use `return`s mid-function.

In the real world, experienced programmers use `return` all throughout a function, e.g., inside an `if` statement or a loop. This is not necessarily a bad thing, and can even be a good practice, if done properly. The problem is that beginner programmers have not yet developed the experience necessary to understand when this can be a bad idea and can lead to bugs, so we strongly recommend (or sometimes require) `return` to be at the end of a function. Even worse, some beginner programmers use `return` throughout a function as a crutch because they are weak at using boolean logic and control structures properly.

Important: the return statement only copies data back. If you think through the process, this is the only thing that makes sense. However, some students get confused about variable names and what gets changed when you return. To keep it simple, remember: only a copy of the data is sent back.

Let's update our example, and show how it could be used:

```
int myMax(int a, int b)
{
      int max = a;
      if (b>a)
      {
            max = b;
      }
      return max;
}

void draw()
{
  int small = 10;
  int large = 20;
  int bigger = myMax(small,large);
}
```

Here, the program starts at the `draw` block as usual. When `myMax` is encountered, it jumps to the `myMax` block, and copies the small and large values into `myMax`'s local variables `a` and `b`. Once `myMax` is done, the `return` statement jumps back to the `draw` block, and copies the `myMax` result back (from the `max` variable) and stores it in `bigger`. Whew!

14.6 **Example: Randomly Moving Video Game Enemies**

We are going to make a program with some squares moving randomly around the screen. Imagine that these are moving-target enemies in a video game, so we'll call them "bad guys." First we'll implement one bad guy, then three, and then update the code to use functions. The purpose of this example is to get practice, and to also reiterate and see how nicely functions can be used to clean up code as your programs grow.

Let's first do one enemy.

Start by setting some globals to define the bad guy's size and color; each bad guy will be different, as we scale up to multiple bad guys. We also want some globals to keep track of the current position. Start this guy at `0,0`. Finally, let's set some finals to define the maximum move and the background color:

```
final int MAX_MOVE = 20;
final int BG_COLOR = 0;
float badGuy1Size = 20;
int badGuy1Color = 255;
float badGuy1X = 0;
float badGuy1Y = 0;
```

Now, in our `draw` block, we need to make the guy move randomly. We need to handle `x` and `y` separately. First, let's do `x`. Since the maximum that the block can move is 20 in either direction, we actually have 40 possible movement positions (20 to the right, 20 to the left). So, we need to generate a random number between -20 and 20. We have done this in earlier examples, so I will not explain here how it works, but you should be able to figure the following code out:

```
float move = random(MAX_MOVE*2)-MAX_MOVE;
```

So `move` can be from -20 to 19.99, effectively 20. To move the bad guy, we simply add this to the bad guy's current `x` location., and use `max` and `min` to make sure it stays on the screen:

```
  badGuy1X += move;
  badGuy1X = min(badGuy1X, width-1);
  badGuy1X = max(badGuy1X, 0);
```

Actually, it goes off the right side until the edge. How can you fix this?

Now, we need to repeat all of the above for the `y` coordinate, including creating a new random move.

```
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1Y += move;
  badGuy1Y = min(badGuy1Y, height-1);
  badGuy1Y = max(badGuy1Y, 0);
```

Finally, let's draw the bad guy. Don't forget to clear the background at the beginning of your draw block.

```
fill(badGuy1Color);
stroke(badGuy1Color);
rect(badGuy1X, badGuy1Y, badGuy1Size, badGuy1Size);
```

So this isn't so bad, but your senses should start tingling when you see that repeated code for moving the `x` and `y` coordinate. Let's now scale this example up to three independent bad guys. Set them to different sizes and colors. Do this on your own.

Now we see a real mess… Here is my code:

```
// bad guy 1
  float move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1X += move;
  badGuy1X = min(badGuy1X, width-1);
  badGuy1X = max(badGuy1X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1Y += move;
  badGuy1Y = min(badGuy1Y, height-1);
  badGuy1Y = max(badGuy1Y, 0);

  fill(badGuy1Color);
  stroke(badGuy1Color);
  rect(badGuy1X, badGuy1Y, badGuy1Size, badGuy1Size);

  // bad guy 2
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy2X += move;
  badGuy2X = min(badGuy2X, width-1);
  badGuy2X = max(badGuy2X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy2Y += move;
  badGuy2Y = min(badGuy2Y, height-1);
  badGuy2Y = max(badGuy2Y, 0);

  fill(badGuy2Color);
  stroke(badGuy2Color);
  rect(badGuy2X, badGuy2Y, badGuy2Size, badGuy2Size);

  // bad guy 3
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy3X += move;
  badGuy3X = min(badGuy3X, width-1);
  badGuy3X = max(badGuy3X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy3Y += move;
  badGuy3Y = min(badGuy3Y, height-1);
  badGuy3Y = max(badGuy3Y, 0);

  fill(badGuy3Color);
  stroke(badGuy3Color);
  rect(badGuy3X, badGuy3Y, badGuy3Size, badGuy3Size);
```

First, we can simplify it by creating a new function for drawing. This takes parameters for the location, size, and color, so the function can be generic and not use globals:

```
drawBadGuy(x, y, size, color)
```

The header for this will take four parameters, and return nothing, so our function header will be

```
void drawBadGuy(float x, float y, float size, int col)
```

Then, inside the function, we just use the local variables to set the colors and draw:

```
void drawBadGuy(float x, float y, float size, int col)
{
  fill(col);
  stroke(col);
  rect(x, y, size, size);
}
```

Now we can simplify our draw block code to use this function for each bad guy instead of repeating the code:

```
// bad guy 1
  float move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1X += move;
  badGuy1X = min(badGuy1X, width-1);
  badGuy1X = max(badGuy1X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1Y += move;
  badGuy1Y = min(badGuy1Y, height-1);
  badGuy1Y = max(badGuy1Y, 0);

  drawBadGuy(badGuy1X, badGuy1Y, badGuy1Size, badGuy1Color);

  // bad guy 2
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy2X += move;
  badGuy2X = min(badGuy2X, width-1);
  badGuy2X = max(badGuy2X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy2Y += move;
  badGuy2Y = min(badGuy2Y, height-1);
  badGuy2Y = max(badGuy2Y, 0);

  drawBadGuy(badGuy2X, badGuy2Y, badGuy2Size, badGuy2Color);

  // bad guy 3
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy3X += move;
  badGuy3X = min(badGuy3X, width-1);
  badGuy3X = max(badGuy3X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy3Y += move;
  badGuy3Y = min(badGuy3Y, height-1);
  badGuy3Y = max(badGuy3Y, 0);

  drawBadGuy(badGuy3X, badGuy3Y, badGuy3Size, badGuy3Color);
```

It's a little nicer, but as you can see there is still a lot of repetition. We need to think about how to put the repeated commands into a function. In this case, there is a challenge: the problem is that our code changes the values inside the `badGuyX` and `badGuyY` variables. If we change those inside globals inside a function, then our function is not generic and we need a special function for each bad guy. Alternatively, imagine we created a command to be used something like this:

```
moveBadGuy(x, y, xmin, xmax, ymin, ymax);
```

which moves a bad guy, currently at `x` and `y`, and makes sure it stays within the minimum and maximum. We could use this, for example, as:

```
moveBadGuy(badGuy1X, badGuy1Y, 0, width-1, 0, height-1);
```

The problem here is that when we call this new function, all our data (bad guy X and Y) is copied into the function. Since it is only copied, any changes we do within that function will not be reflected back in our `badguy` variables. The only way to get information back from this function is to return it.

Unfortunately, a function can only return one piece of data, and we want a function to update both the x and y components of the bad guy. This means we cannot have a function that changes both the `badGuyX` and `Y` at the same time.

Let's reconsider the approach – notice how the movement for the `x` and the `y` are very similar. It generates a random number, it adds it to the `x` or `y`, and then checks to make sure it is not too big or too small.

In this case, the differences between the `x` and `y` cases are:

- We start with the X **or** Y coordinate
- We cap the X **or** Y to the width or height
- We store in the X **or** Y

Everything else stays the same. What if we created a function that worked on only one coordinate at the time? Something like…

```
float doMove(float position, float minimum, float maximum)
```

For example, then we could rewrite the above movement code to:

```
// bad guy 1
badGuy1X = doMove(badGuy1X, 0, width-1);
badGuy1Y = doMove(badGuy1Y, 0, height-1);
```

Assuming that the `doMove` function takes a position, adds a random move to it, caps it to safe screen bounds, then this is a great simplification!! So, let's try to make the `doMove` function. First, the header takes the three parameters, all of them are floating point, and returns the floating point result.

```
float doMove(float position, float minimum, float maximum)
{
```

Then, we use these parameters inside the `doMove` block to do the work. Remember, we use these as local variables and completely ignore what names or variables are used in the draw block that calls this function (badGuyX, etc.) as all the values are copied in:

```
float doMove(float position, float minimum, float maximum)
{
  float move = random(MAX_MOVE*2)-MAX_MOVE;
  position += move;
  position = min(position, maximum);
  position = max(position, minimum);
  return position;
}
```

The return statement is absolutely necessary!! Remember that when the function is called, the data is only copied in. The changes made inside the function do not reflect back in the original code, since only the copies are changed. In this case, the changed copy needs to be copied back to the main code.

There is one more thing to notice here. This function only uses globals that are final and constant – all the changing data comes in via the parameters. This is good practice and a good habit to get into.

Using this new function we can now re-write that draw code more simply. I include the complete final program below.

Even though we now have two additional functions, the draw code becomes a lot easier to wrap our heads around. Repetitive code is kept to a minimum since it is factored out to the functions. Scroll back and look at the initial example, how long and repetitive it was, to compare.

Here is the final program. Notice that although, overall, it is still pretty long, each section – trying to solve a specific problem – is much smaller and more easily digestible. You are able to focus on one aspect at a time instead of solving everything at once.

```
final int MAX_MOVE = 20;
final int BG_COLOR = 0;
float badGuy1Size = 20;
int badGuy1Color = 255;
float badGuy1X = 0;
float badGuy1Y = 0;
float badGuy2Size = 40;
int badGuy2Color = 100;
```

```
float badGuy2X = 0;
float badGuy2Y = 0;
float badGuy3Size = 5;
int badGuy3Color = 180;
float badGuy3X = 0;
float badGuy3Y = 0;

void setup()
{
  size(500, 500);
}

void drawBadGuy(float x, float y, float size, int col)
{
  fill(col);
  stroke(col);
  rect(x, y, size, size);
}

float doMove(float position, float minimum, float maximum)
{
  float move = random(MAX_MOVE*2)-MAX_MOVE;
  position += move;
  position = min(position, maximum);
  position = max(position, minimum);
  return position;
}

void draw()
{
  background(BG_COLOR);

  // bad guy 1
  badGuy1X = doMove(badGuy1X, 0, width-1);
  badGuy1Y = doMove(badGuy1Y, 0, height-1);
  drawBadGuy(badGuy1X, badGuy1Y, badGuy1Size, badGuy1Color);

  // bad guy 2
  badGuy2X = doMove(badGuy2X, 0, width-1);
  badGuy2Y = doMove(badGuy2Y, 0, height-1);
  drawBadGuy(badGuy2X, badGuy2Y, badGuy2Size, badGuy2Color);

  // bad guy 3
  badGuy3X = doMove(badGuy3X, 0, width-1);
  badGuy3Y = doMove(badGuy3Y, 0, height-1);
  drawBadGuy(badGuy3X, badGuy3Y, badGuy3Size, badGuy3Color);
}
```

14.7 **Common Function Pitfalls**

Function are a core part of computer programming and you will use them from now on, ubiquitously, in your programming career. Once you get the hang of them they will be really simple. In the meantime, there are a few hiccups that often catch people.

The key one is remembering that ***the data is only copied***. Some people think that if you use a variable in a function call, like…

```
int maxSize = 50;
drawRandomSquares(maxSize);
```

Then somehow what happens inside the function can change the `maxSize` value. The code above *guarantees* that `maxSize` still has size 50, no matter what happens inside the function call.

Sometimes, to make things more confusing, the function will have the same parameter name as the variable you are using. For example:

```
void changeData(int data)
{
 data = data * 2;
}
void draw()
{
 int data = 4;
 changeData(data);
 println(data);
}
```

Here, in the `draw` block, we call `changeData` by passing the `data` variable as the parameter. Even though `changeData` has a parameter called `data`, because of scoping, these are completely different variables. Even though `data` changes inside the function, the original `data` remains unchanged. The `println` output is 4, regardless of what happens inside the function.

Let's break this down. The function `draw` runs. Creates a local variable called `data`, and sets it to equal 4. Then it calls the `changeData` function, and copies the information in `data` (4) to it. Only the number 4 is copied, the function has no clue what variable that data came from.

Inside the `changeData` function, a new local variable is created called `data`. This coincidentally has the same name as the variable in `draw`, but since they are in different blocks (in different scopes) they are completely unrelated. The data passed to the function (4) is copied into our new local variable, `data`. Inside `changeData`, we multiply this by 2, and set the `data` variable to 8. We return to the `draw` function, but no data is returned. The local variable in the function, `data`, is destroyed and the data lost.

Since the information in the `data` variable in `draw` was only copied, the `data` in our `draw` block didn't change. It is still set to 4. The `println` prints out 4.

This can be quite confusing. The best way to get this is to type up the program and tinker with it.

Another point about the above example, is that some people think that the parameter name in the function call must match the name in the function header. Here, the variable in the `draw` block could be called anything, and the code would work the same. The same name used, `data`, is just a coincidence.

The function scope rules also explain another case:

```
int myMin(int a, int b)
{
  int result = a;
  if (b<a)
    result = b;
  return result;
}

int myMax(int a, int b)
{
  int result = a;
  if (b<a)
    result = b;
  return result;
}
```

Here, notice how two function use the same variable names for parameters. Is this okay? Yes – because of local scope. Each function has its own scope, so can have variable names. Outside of the variable scope, it cannot be viewed or accessed, so you don't need to worry about it.

A trick for dealing with these issues is to ***keep tunnel vision***. When you are working on a function, look at your local parameters only and use that information to do what work you need to do. Forget the rest of the program. Don't look at how other code is calling the function. Likewise, when you are using a function, just be sure that you know what it does, and that you are passing the correct data to it. Don't worry about what exactly is going on (or what the parameters and variables are called) inside the function. For example, you don't know what happens inside the line command but you use it all the time! Whoever wrote the `line` command, implemented it without knowing (or caring…) about exactly how you will use it!

14.8 **Functions Calling other Functions: Palindrome Tester**

Any function can call any other function, this is not just restricted to the `draw` block. In fact, your `draw` block is just a function that processing is calling repeatedly. Inside the `draw` function you can call your other functions. Inside your other functions, you can call even more! This can go on and on (and does!). In fact, a function can call itself (or function A can call function B, which then calls function A), a technique called recursion. This is a hard topic, however, and one beyond the scope of this course.

Advanced: When a function calls another, it saves all the important information – where it came from and where to go back to, as well as the parameters for the function, on a data structure called a stack. You can imagine the stack like a stack of old books, and each time a function is called, that information gets put on the stack; another book gets put on the pile. When the function is done and returns, the last item on the stack is removed, telling the program where to go back to. If you use recursion, and a function calls itself, and it loops too deeply, your stack gets too large (all those memories of where to go back to), and you get an error called *stack overflow*. It can be challenging to avoid those, so we'll look at recursion in a different course.

Let's work through an example with nested function calls: a program to test if a string is a palindrome.

A palindrome is a word that, if you reverse it, gives you the same word. For example "`tacocat`" reversed gives you "`tacocat`". To test if a word is a palindrome, what we need to do is to first reverse the word, and then compare it against the original.

In this example, we will take a top-down approach to developing our solution. That is, we will work from the `draw`

loop first, and create functions as we go.

The particular technique employed in this example is very common in programming. Sometimes, you want a task done, but haven't quite figure out how to do it yet. So, what you do is to offload the problem until later: create a fake function and pretend it does what you want, and use it in place to solve your current problem. Then, you come back and implement that function later. This helps you focus on one problem at a time and minimize how much you get overwhelmed. To make this concrete, let's make our `draw` block call some new function `isPalindrome` to see if some text is a palindrome:

```
void draw()
{
  background(0);
  String test = "racecar";
  if (isPalindrome(test))
  {
    text(test+" is a palindrome", 10, 250);
  }
}
```

At this point, `isPalindrome` doesn't exist – we just made it up. In this case, you can see that the function takes a `String` parameter, and returns a `boolean` result – true if the string is a palindrome, and false if not. We created this fake function because it lets us focus on our `draw` block – we test some string, and if it's a palindrome, put some text on the screen. Otherwise, do nothing. Now we can leave the `draw` block alone and move on to the next problem.

Because we made this fake function, this won't compile yet. We now need to implement `isPalindrome`. First, put in a ***stub*** or ***skeleton*** of the function, to make your program run:

```
boolean isPalindrome(String test)
{
  return false;
}
```

We do no work and return garbage data, just to make the program run. It should run – but not work – now.

Once you write up the function header, you should forget about how the function is used; all you need to know is that it takes a string, and returns true if it's a palindrome, and false otherwise. This is an example of the tunnel vision discussed earlier.

To test if a string is a palindrome, we first reverse it, and then compare to the original. This is two separate problems, so let's focus on one at a time. Let's assume we have some function to reverse a string (let's make one up: `reverseString`) that takes a `String` parameter and returns the reversed string. If we have that, then palindrome checking is easier:

```
boolean isPalindrome(String test)
{
  String reverse = reverseString(test);
  boolean match = (reverse.equals(test));
  return match;
}
```

Here, we reverse the string, and compare it to the original, returning the result. Again, create a stub for your new `reverseString` function to get your program to run. It still won't work correctly, however.

```
String reverseString(String original)
{
  return "";
}
```

Again, we return garbage to just get it run. Here I randomly chose the empty string.

Reversing a word is a little tricky. The way we do it is to use a for loop to pick out each character one by one, and then put the string back together in the opposite order.

We can make a `for` loop to go through this string character by character. However, where do we put the new characters? We need a new string to put them into. Let's call this `reverse`. Then, the `for` loop goes through each character, and adds them to `reverse`.

```
String reverse;
for (int i = 0; i < original.length(); i++)
{
  char c = original.charAt(i); // grab char i from original
  reverse = reverse + c; // toss it onto the reverse
}
```

There are two problems in this code. Can you spot them?

The first problem is that the variable `reverse` needs an initial value. In our code, the first time we use `reverse`, we *add* `c` onto it, but since we never originally set it to a value, Processing gets confused – we can't add to it if the value is not yet defined. What would be a reasonable initial value? We can set this to the empty string when we create it.

```
String reverse = "";
```

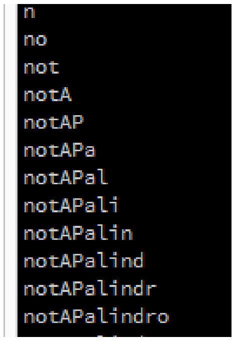
The other problem is not so obvious. When we write code like this, it is always a good idea to think about how you can test if it's working properly. In this case, let's toss the strings to console to see if the reversing works. After the for loop:

```
println(original);
println(reverse);
```

If you run this code, you can see that the two strings printed are the same! The reverse didn't work.

It can be confusing to see why this is, so we need a debugging strategy here. Inside the for loop, let's insert a `println` statement to show us what is happening to the `reverse` variable as it gets created:

```
println(reverse);
```



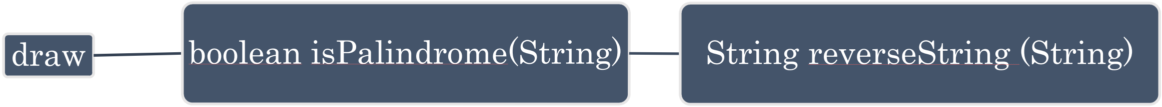
```
n
no
not
notA
notAP
notAPa
notAPal
notAPali
notAPalin
notAPalind
notAPalindr
notAPalindro
```

If you run this now, and look at the console, you can see that the reverse actually just becomes a copy of the original string. The reason for this is because we are taking the characters from left to right from the original string, and adding them on the *end* of the new string. To reverse it, we instead need to add the characters onto the beginning:

```
reverse = c + reverse;
```

Finish up by making sure to return the appropriate value from your function, and now your program should work as expected! Try different strings, and test if they are palindromes.

Great! We created functions that call other functions. We kept tunnel vision and implemented only one section at a time, simplifying our mental load. The following flow chart shows how the functions called each other.



14.9 **Best Practices – global and local variables**

The way we use Processing so far brings up an issue when using functions: we have both global and local variables.

You should avoid using global variables in a function as much as possible, except for final constants. If you start modifying global variables in a function it becomes hard to keep track of and it doesn't scale well to larger programs. For example, you may forget that you modify the global in one function, and this can make another function work incorrectly.

You are much better off thinking of functions as stand-alone as often as possible. This will help you better get a grasp of how functions are properly used, and how data is sent to and returned from them.

If there is some changing, specific data (like, what color to use, where to draw a bad guy, etc.) that can vary whenever the function is called, then pass that information to the function as a parameter. A function should have all the required information to do its job from its parameters as much as possible, and from any global constants (finals).

You have to be very careful with side effects!! Like that time that I took.. nevermind.

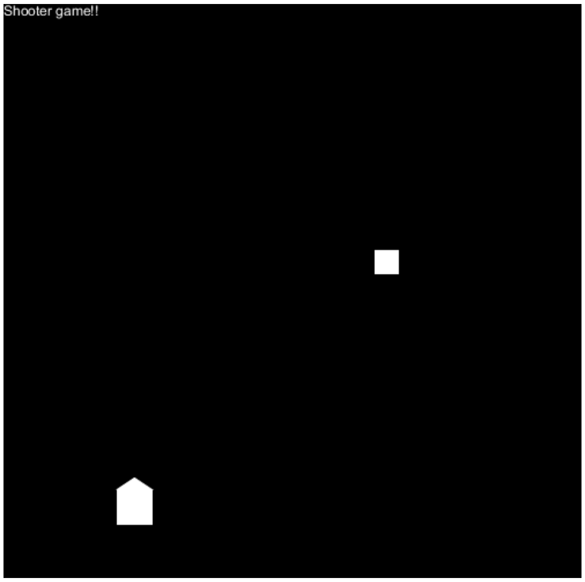
Functions likewise should not change any global information and should return the results through the return mechanism. If a function changes other values this is called a ***side effect*** of the function, and is generally considered to be higher risk. The problem is that when you use a function with side effects, they may not be obvious and you may have forgotten about them, which leads to bugs. Hilarity ensures (or rather, hours of banging your head against the wall debugging).

This relates to the idea of keeping tunnel vision. As much as possible, when implementing functions, try to make it independent of the rest of the program. If you can avoid using globals – and use parameters instead – then do it, as it makes it easier to implement. Keeping an entire program in your head at once is impossible so we need techniques to simplify the problem into smaller chunks. If we have a good function, then implementing the function lets us solve one simple problem while ignoring the rest of the program. For example, given:

```
double addTax(double price) {
      //…
      return?
}
```

You know you have a `double` value (price) and need to return a `double`, all you need to do is think about how to add the tax. You don't need to know how this function will be used.

14.10 **Example: Top-Down Programming for a Space Shooter**



Let's make a shooter video game with one bad guy. The space ship is linked to the mouseX (so that it moves left and right but not up and down). If the mouse button is pressed, the ship fires a bullet, but only one bullet is out at a time. The bullet goes up the screen until it either hits the bad guy or goes off the edge of the screen. If the user clicks while a bullet is flying, nothing happens.

In this example, let's try to make our functions completely stand-alone, and not use any globals at all – everything that the functions need should come in as parameters. If you do the related exercise at the end of the chapter, you will really see how this is a good idea.

We will approach this with four steps:

1. Write the program in draw as a series of steps in comments, in English.
2. Create fake functions, and use them in draw to get our work done
3. Create empty functions, stubs
4. Start implementing the functions

Using to-down programming, first plan the steps of the program in English and make sure it makes sense on that level.

- draw a title at the top of the screen
- move and Draw the player's ship
- check if a bullet should be shot, and if so, shoot it
- move and Draw the bullet
- move and Draw the bad guy
- check if the bullet hit the bad guy, and if so, stop the game
- draw "WIN" if the game is over.

Why is it bad "guy", isn't that sexist??? I'm a guy, and I'm not bad!!

Whew! Did I miss anything? **Point:** We are thinking through our program without doing any computer programming at all. We are just planning things out.

If we want to avoid using globals in our user-defined functions, then we need to keep all the game logic in the draw loop, since this is the only place we try to modify globals.

Type the above comments into your draw loop. At this point, my program looks like this:

```
// Game details
final int BG_CLR = 0;

void setup()
{
  size(500,500);
}

void draw()
{
  background(BG_CLR);
  //draw a title at the top of the screen
  //move and Draw the player’s ship
  //check if a bullet should be shot, and if so, shoot it
  //move and Draw the bullet
  //move and Draw the bad guy
  //check if the bullet hit the bad guy, and if so, stop the
     // game
  //draw "WIN" if the game is over.
}
```

Now we start trying to turn our comments into computer code. I like to start with the easy stuff, which in this case, is drawing a few strings on screen: drawing the title and the win message (if the game is over). At this point, I don’t want to worry about how I’m going to do it, so I’m going to make a new function that I’ll manage later, to help me. I’ll make one called `drawMessage` that takes a message, a location, and a color. This enables me to forget about those details for later and think about my `draw` logic. Drawing the title is easy. I’ll also make up some globals (give them reasonable values):

```
// draw title
drawMessage(TITLE, TITLE_X, TITLE_Y, TITLE_CLR);
```

The second draw is a little trickier, since I only draw the end-game message if the game is over. I’ll create a `boolean` to tell me if the bad guy is dead, and only draw if that’s true. Now my `draw` block looks as follows. Make sure to create the proper global variable and the constants:

```
void draw()
{
 background(BG_CLR);
 // draw title
  drawMessage(TITLE, TITLE_X, TITLE_Y, TITLE_CLR);

  //move and Draw the player’s ship
  //check if a bullet should be shot, and if so, shoot it
  //move and Draw the bullet
  //move and Draw the bad guy
  //check if the bullet hit the bdad guy, and if so, stop the
  //game

  //draw “WIN” if the game is over.
  if (badGuyIsDead)
    drawMessage(WIN, WIN_X, WIN_Y, WIN_CLR);
}
```

Now, create the stub for `drawMessage`, and make sure the program runs, even though nothing will happen.

At this point, there are two ways we can go: we can implement `drawMessage` now and see if it works, or, we can keep working on `draw` to see if we can figure out the other pieces. I’m going to keep working on `draw`.

Now, let’s move down the list. Moving the player is easy – the `y` coordinate is fixed at the bottom of the screen, and the `x` coordinate is linked to the mouse. Make a global constant for the `y` location, and a variable for `x`, and set `x` to the `mouseX` in the draw block. Drawing is a little more annoying, because of that funny shape (see the picture), so let’s make a fake function and handle it later. To draw the ship, we need the ship’s location, size, size of the nose (see the pic), and the color, so we can make a function that takes those things. Here is my code:

```
// move and draw player ship
playerX = mouseX;
drawPlayerShip(playerX, PLAYER_Y,
               PLAYER_SIZE, PLAYER_NOSE, PLAYER_CLR);
```

Also, make the stub for `drawPlayerShip` so that the program runs.

The next step is tricky. We first need global variables to keep track of the bullet's location. We also need constants for the bullet color, and how fast the bullet should move. Finally, we need to keep track of whether a bullet is currently being shot or not – remember, we can only have one at a time – so we can use a boolean for this. Once we have variables setup, we can start code.

At this point, thinking of whether a bullet should shoot takes some boolean logic, and it's a little confusing, so let's make a new function called `checkShoot` that gives us a boolean to tell us if we fire a new bullet. We just tell it if a bullet is already moving or not. If we should shoot, then we move the bullet to the player's location, and set our "moving" flag to true. This is a little tricky, so try thinking through it. Here is my code:

```
// check if we should fire a bullet, and if so, shoot it
if (checkShoot(playerBulletMoving))
{
  playerBulletMoving = true; // it's alive!!
  playerBulletX = playerX;
  playerBulletY = PLAYER_Y;
}
```

Also, implement the stub for `checkShoot` – it takes and gives a boolean. For now, return a garbage value so that the program runs.

Moving forward, now we move and draw the bullet. Again, create new functions to avoid thinking about the details, and instead focus on the high level game logic. Here, if we have a moving bullet, then we call `moveBullet` with to move it: give it the current `y` and the speed, and it gives us the new `y` value back. After the bullet is moved, we should also check to see if it ran off the top of the screen – if so, kill it (set our flag to false). Finally to draw the bullet, let's make a function called `drawBullet` that takes the position and color, and draws it for us.

```
//move and Draw the bullet
if (playerBulletMoving)
{
  playerBulletY = moveBullet(playerBulletY, -BULLET_SPEED);
  if (playerBulletY < 0) // end of screen
      playerBulletMoving = false;
}
drawBullet(playerBulletX, playerBulletY, BULLET_CLR);
```

We do something similar for moving and drawing the bad guy. You have some flexibility here, as to how you want it to move. I'm going to use the random move from our earlier example in this chapter. Think through what variables and globals you need. Also, as we already have a flag telling us if the bad guy is dead, we should check that and not move a dead bad guy!

```
//move and Draw the bad guy
if (!badGuyIsDead)
{
  badGuyX = doMove(badGuyX, BAD_GUY_MOVE, 0, width-1);
  badGuyY = doMove(badGuyY, BAD_GUY_MOVE, 0, height-1);
}
drawBadGuy(badGuyX, badGuyY, BAD_GUY_SIZE, BAD_GUY_CLR);
```

We only have one piece left – check if the bullet hit the bad guy. Again, let's offload the hard stuff (checking if the bullet hit the bad guy!) into a function. I created `checkHit` that takes the bad guy location and size (as its square) and the bullet location. It returns true if the bullet is inside the bad guy, and false otherwise. If true, we set our dead flag to true.

```
//check if the bullet hit the bdad guy, and if so, stop th
//game
if (checkHit(badGuyX, badGuyY, BAD_GUY_SIZE,
    playerBulletX, playerBulletY))
    badGuyIsDead = true;
```

Whew! We turned that English into high-level game logic. We thought about what work needs to be done – what needs to be drawn, what conditions checked – and what logic we need to make our game happen – if a bullet is flying, if the bad guy is dead – and made a good first pass on the high level game logic. By using user-defined functions, we were able to leave some of the annoying logic (like drawing the player's ship) behind to solve later, and focus on the game problem.

Before going forward, make sure that your program runs – nothing will happen, but it shouldn't raise errors either. This is important, because you want to be able to test pieces as you build them.

I will give much less detail on implementing the actual functions, as you should work on this on your own. The `drawMessage` function should be straight forward, and the other draw functions, `drawPlayerShip`, `drawBullet`, and `drawBadGuy` just take a little bit of tricky geometry at worst. `checkShoot` is simple boolean logic (returns true if the mouse is pressed and we don't already have a bullet), `doMove` is based off our earlier randomly moving square example, and `checkHit` we saw in the boolean chapter: test if a point is inside a box.

Your strategy here should be piece wise: implement a function, test the behavior, and move on. You will surely encounter bugs, but try to fix whatever you can before moving on. An easy way to do it is to implement all the drawing functions first (so you can see the player), then the enemy move (so you can see him moving around), and then the bullet firing. Finally, test if the bullet hits the enemy.

14.11 **Example: Dice Game**

Let's do another quick example, this time with less hand holding. We will make a gambling dice game simulator to test out how much you may win in a game. Here are the pieces

- Throw a six sided dice
- Update your bank balance based on the roll
    - 1 -> you win 10% of your money (multiply by 1.1)
    - 2 -> you win 20% of your money (multiply by 1.2)
    - 3 -> you win 30% of your money (multiply by 1.3)
    - 4 -> you win 40% of your money (multiply by 1.4)
    - 5 or 6 -> you lose 35% of your money (multiply by .65)
- Keep playing until you either fall below $2 or go above $1000. Start with $10.
- Play one round per `draw` loop.

Try to spend some time thinking through how this should work as a program. As a starting point, here is my English version.

- If we are still playing
    - Throw a die
    - Update our balance based on the result
    - Check if we should stop
- Display statistics (how many rolls, and our current balance)

Let's first put a skeleton of the draw and then consider what data needs to pass around

```
void draw()
{
  background(BG_CLR);
  if (!finished)
  {
    throwDice( ??? );
    updateBalance( ???);
    shouldStop( ???);
  }
  displayStats( ???);
}
```

The throw dice function probably doesn't need any data, as the number of dice sides, etc., should be fixed as a global final. However, it does give us an integer (a roll) that we need to store.

```
int roll = throwDice();
```

To update the balance, we need to tell the function our current balance, AND, what the roll was, so it needs two pieces of data. Further, it will pass us back the updated balance, so we should store that as our new balance.

```
balance = updateBalance(balance, roll);
```

In order to determine if we should stop or not, we need only to check the balance. Also, we need to store the result. We already have the boolean `finished` – let's update that based on the result.

```
finished = shouldStop(balance);
```

Finally, we need to display the statistics. It will need our current balance. Also, it will need to know how many rolls we simulated, so we need a new counter variable that updates every time we do a roll. Be careful – make sure to update this inside the `if` statement (if not finished), since once done, we will still display stats but not do more rolls.

```
displayStats(balance, rolls);
```

Here is my final draw block:

```
void draw()
{
  background(BG_CLR);
  if (!finished)
  {
    int roll = throwDice();
    balance = updateBalance(balance, roll);
    finished = shouldStop(balance);
    rolls++;
  }
  displayStats(balance, rolls);
}
```

Now we can implement the functions based on the headers created here. These are pretty straight forward, so try doing it yourself.

Check your Understanding

14.12 Check Your Understanding: Exercises

Exercise 1. Create a function that helps calculate compound interest: it takes three parameters, `principle`, `rate`, and `years`. Use floating point numbers, but remember that this would not be acceptable for real money. Calculate the final amount by adding the rate to the principle once per year, for the given number of years. You can use a loop for this, or look up the `pow` command for a short cut. Return the amount as a floating point.

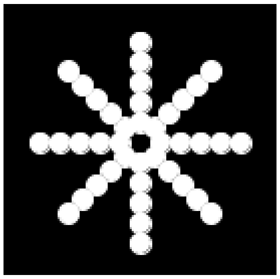
Exercise 2. Make a program that has two user defined functions: `roundUp` and `roundDown`. These take a single floating point number and return an integer. To round, try to do it yourself with logic and a cast, or, look up the ceiling and floor functions.

a. Make a new function called `roundEven` that takes a floating point number as a parameter, calls `roundUp` and `roundDown`, and then returns whichever integer result is even. In the case where neither is even, return the next even number larger than the one given.

Exercise 3. Update the shooter game example (14.10) to have three bad guys! This should really highlight why having stand-alone functions that do not modify globals is a good idea.

Exercise 4. In this exercise, you will draw a “waving octopus” graphic at the mouse location.

a. Draw `NUM_ARMS` “arms”, each consisting of a line of `CIRCLES_PER_ARM` small circles, with diameter `CIRCLE_SIZE`. The values 8, 5, and 10 are used in the image on the right. Make the arms stick straight out from the mouse position, not curved as they are shown here. The center of each circle in an arm should be a distance of `CIRCLE_SPACING` from the centers of the adjacent circles in the same arm (or from the mouse position, in the case of the closest circle). Do this by writing the following two small functions, exactly as

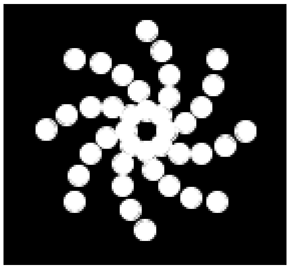


specified:

`void drawCircle(float theta, float distance, float xc, float yc)` – this function draws one circle diameter `CIRCLE_SIZE` at `distance` away from the point `(xc,yc)`, in the `theta` direction. For example, `drawCircle(PI/2, 100, 50, 50)` should draw a circle centered at `(50,150)` – straight down 100 pixels from `(50,50)`.

`void drawOctopus(float x, float y)` – use two nested `for` loops to draw an "octopus" with its center at `(x,y)` using the global constants. One loop goes through each arm, and the other through the ring. Use `drawCircle` to draw all circles. The arms should stick out at evenly-spaced angles in a circle around `(x,y)`.

Call `drawOctopus` from `draw()` to draw an "octopus" at the mouse location. You should see a kind of star, with arms straight out, from the mouse.

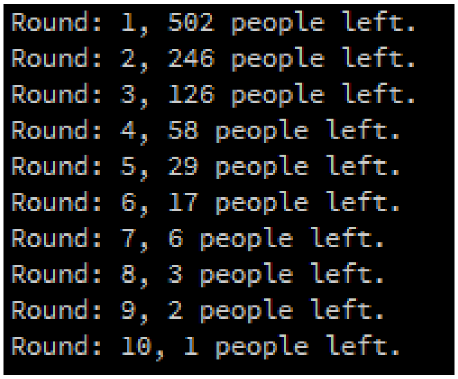


b. Make sure the above works before attempting this one. Make the arms "wave" by using a slightly different angle (theta) for each of the circles that make up an "arm", so that the arm curves, instead of sticking straight out. Change the `drawOctopus` function to be `void drawOctopus(float warp, float x, float y)`. Add `warp` to the angle of the first (closest) circle in an arm, `2*warp` to the angle for the second one, `3*warp` to the angle for the third one, etc. If `warp=0` then the arms will be straight, as before, but larger or smaller values for warp will make the arms curve to the right or left. (They will no longer be exactly distance apart, but that's OK.)

Change the `draw`() function to use a different warp value from one frame to the next. Start with 0, and every frame, a value `warpSpeed` should be added to the amount of warp, until it reaches `WARP_LIMIT`. Then the warp value should start to decrease, going back to 0 and continuing down to `-WARP_LIMIT`, at which point it should start to increase again, repeating the cycle forever. Hint: Note that `warpSpeed` is a variable, not a constant, so that its value can change from positive to negative and back again.

Exercise 5. We humans are really bad at understanding probabilities, odds, and how they scale up with populations. We all have some anecdotal story of something amazing that happened to us or a relative, or some magical cure for ailments (Last time I had a cold, I drank ginger tea, and I got better right away!! Ginger tea must cure colds!).

Let's do a simulation of a population of people rolling dice. We'll start with a *2* sided dice to simulate a coin toss. If I asked you how likely you are to get, say, 10 heads in a row, you know that it's very unlikely. However, if only a thousand people around the world do it, someone will likely do it just by random chance.



```
Round: 1, 502 people left.
Round: 2, 246 people left.
Round: 3, 126 people left.
Round: 4, 58 people left.
Round: 5, 29 people left.
Round: 6, 17 people left.
Round: 7, 6 people left.
Round: 8, 3 people left.
Round: 9, 2 people left.
Round: 10, 1 people left.
```

Imagine we lined up 1000 people and asked them all to toss a coin. If you get heads, stay, tails leave. On the first toss, we expect about 500 people left. Do it again, on the second toss, about 250 people left. And so on. Eventually, there will be 1 or 2 people left, and those people will feel amazing- they tossed so many heads in a row!

First, let's setup the program. One nice trick is that we can put the command `noLoop()` in the setup block, and then the draw will only execute once:

Our `draw` block will do the following

- Start with an initial number of people (say, 1000)
- While there is still more than 1 person left
  - Call the `doARound` function with the number of people you have, and save its return as the number of people left.
  - Increase the number of rounds you ran by 1.
  - Call the `printUpdate` with your people count and rounds to display the status

That's it! Your `draw` block will loop while people are left, and, count how many rounds it took.

You need to implement the following methods:

`int diceRoll(int sides)` – takes the number of dice sides as a parameter, and returns a random dice roll, where 1 <= roll <= sides.

`void printUpdate(int people, rounds)` – uses `println` to give the current status, as in the image.

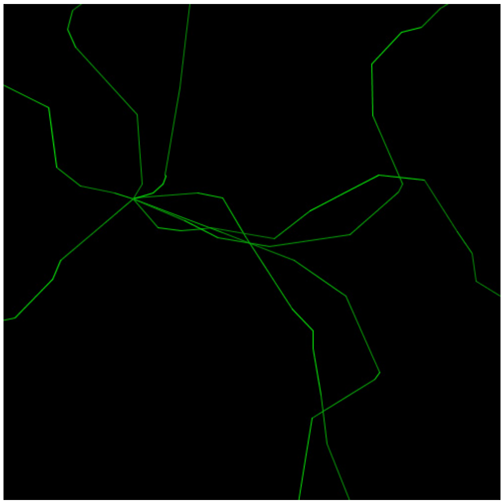
`int doARound(int people)` – takes the number of people left, and simulates a round of dice rolling:

- Use a variable to remember how many people to cut from the pool
- Use a for loop to go through each person
  - Roll a 2 sided dice
  - If the dice number is NOT 1, cut the person.

- Return the new number of people – old number minus the cut number.

This will tell you how many rounds you get until you have 1 or fewer people left. In other words, how many rounds did some people last, getting a 1 (say, a heads) each time? Try changing the parameters – change the 2-sided dice (a coin) to a 6 sided dice. Try changing the number of people. Does your intuition work?

Exercise 6. For this lab you make lightning come out of your mouse!! (just like the Emperor in Star Wars). You will need functions – to make things simpler – and a while loop. The while loop is useful because, since each stroke of lightning is random, you don't know how long it'll take to get to the edge of the screen.



You will use the following global variables: `FLASHES` (how many strings of lightning, the picture has 8), `MAX_PIECE_SIZE` (the longest lightning segment before it shifts direction again, `MAX_SHIFT` (the maximum angle it changes direction on each segment).

Your draw loop uses a for loop to call your function `drawLightning` `FLASHES` number of times. Each time, you call `drawLighning` with the mouse position.This way, you'll get `FLASHES` number of lightning streaks from the mouse.

Implement the following functions:

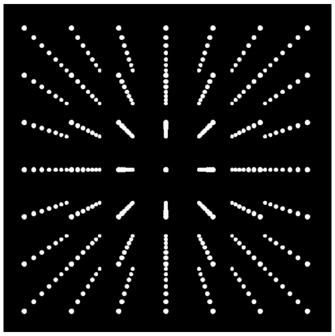
`shiftAngle` – takes an angle, and the maximum shift amount as parameters. Returns the new, shifted angle. Use `MAX_SHIFT` to generate a random shift, in either the clockwise or countreclockwise direction, and add it to angle. Return the new angle.

`onScreen` – takes an `x` and a `y`, and returns true of that coordinate is on the screen, false otherwise.

`drawLightning` – takes a start position and draws a stroke of lightning to the edge of the screen.

- Start at a random angle
- While the `x,y` is on the screen (call `onScreen`)
  - Calculate a random lightning segment size within the valid range
  - Use `cos` and `sin` with your angle and new size to determine the end point of the segment. Remember to add the current position to the result.

- ➢ Draw a line from `x,y` to the `endpointX` and `endpointY` (use a random green color if you want! ☺)
- ➢ Set your new `x` and `y` as the end point you calculated, for use next time
- ➢ Update the angle using shiftAngle.

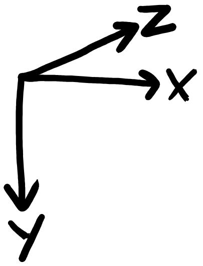


Exercise 7. 3D! Current computer screens are not capable of doing real 3D – they're flat! Anything "3D" you see on a computer screen is an illusion – completely faked. There are a lot of great ways to fake 3D, and to trick people's eyes into seeing 3D. One of the most fundamental ways is by simulating perspective:

- Objects that are further away appear to be smaller
- Objects that move further away appear to converge.

Look at the railroad image. Because of perspective, the tracks appear to get closer together as they get further away. The point where they appear to meet each other in the distance is the "vanishing point". Also, the trees in the distance are not much bigger in the picture than a single railroad tie in the foreground. Our brains pick up on these cues and we know that this is actually a 3D scene because we know that the tree is bigger than the tie.

We will draw a 3D grid of "spheres" (circles when drawn in 2D), as shown in the image.



We have used "flat" 2D `(x,y)` coordinates a lot. To use 3D, we need to add a z coordinate to get `(x,y,z)`. Look at the diagram to the right. Our usual `X` axis goes to the right across your screen, and the `Y` axis goes downward on your screen. The new `Z` axis goes straight into your screen, away from you.

These are virtual coordinates – they're not measured in pixels, and our screen does not really have a `z` axis, so what we need to do is **project** our virtual 3D points `(x,y,z)` onto normal 2D canvas points `(px,py)`. ("`px`" means "projected x"). Note: Processing has a built-in way to handle 3D points, but we won't use that here. You can investigate Processing's 3D capabilities on your own, if you're interested.

The basic idea behind perspective projection is to divide the x and y coordinates by the z coordinate to get px and py. That way, the larger the z coordinate is, the more px and py will decrease, and appear to move toward the "vanishing point" (0,0) – as the railroad tracks did.

But if we just divide by z itself, our px and py coordinates will get *very* small *very* quickly and you'll only see a cluster of dots at the center of the screen. Use the constant PERSPECTIVE (set at 0.002) to control the amount of perspective (how fast px and py converge as z gets larger). Divide x and y by PERSPECTIVE*z instead of just z to get px and py. Object sizes (such as the diameter of a circle or sphere) also must be divided by PERSPECTIVE*z in the same way, since objects also must appear smaller when they are farther away.

Note that perspective can make things shrink, as well as get bigger as they get closer (because PERSPECTIVE is a very small number). Our illusion works best when we look at objects far away "into the screen", not directly in front of our noses, and so we'll want z values that are large compared to x and y.

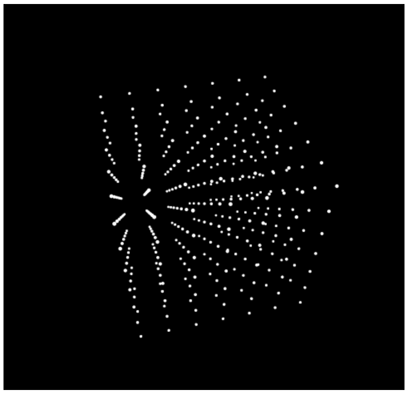
a. Write a function void drawProjectedCircle(float x, float y, float z, float diam) which draws a "sphere" (a circle, really) with a center at the 3D point (x,y,z), and a diameter of diam. Use the projection method described above to modify x, y, and diam before drawing the circle.

   There is one problem: if you happen to get z=0, you will divide by 0 (things are so close that they appear to be infinitely large – they're right in the middle of your eyeball! And z<0 is even worse – beyond infinity?) You need what is called a clipping plane in graphics. Your drawProjectedCircle function shouldn't attempt to draw (or even do the calculations) for any point unless z>0!

b. Write a function void drawDotGrid(int minValue, int maxValue, int spacing, float diam), to draw a three-dimensional grid of circles. We usually use two nested for loops to generate 2D grid, but here, use *three* nested **for** loops to generate iterate over a range of (x,y,z) points in a grid, from minValue and maxValue, spaced spacing units apart. For example, drawDotGrid(-30,30,10,1) should draw circles of diameter one on a grid spaced 10 apart, from -30 to +30 (a total of 7*7*7 = 343 points). Use drawProjectedCircle to do all of the drawing. Add mouseX do the z coordinate so that the mouse can be used to zoom in.

c. Once you get it working, add an offset so that the "vanishing point" (0,0) is moved to the center of the canvas.

Exercise 8. Update exercise 7 to have 3D rotations controlled by the mouse – now you're playing with power!



First, add a constant "zoom factor" of `150` to all z co-ordinates, instead of `mouseX` as was done above. We'll need the mouse to do other things, and `150` will "push" the cube just far enough "into the screen" to give it a nice size. You could experiment with other values, too. (You *were* going to use a named constant, right?)

To rotate a 2D point `(x,y)` an angle of θ radians around the center of rotation `(0,0)`, giving a new point `(rx,ry)`, the math is not very complicated (`rx` is short for "rotated x"):

$$rx = xcos(\theta) + ysin(\theta)$$

$$ry = -xsin(\theta) + ycos(\theta)$$

To rotate a 3D point `(x,y,z)` around the `X` axis (think of grabbing the `X` axis and spinning it), just leave the `x` coordinate unchanged, and use the above formulae with `y` and `z` instead of `x` and `y`. Similarly, to rotate the point around the Y axis, use the formulae with `x` and `z`. Using the formulae as they are above will spin around the Z axis. Try that first.

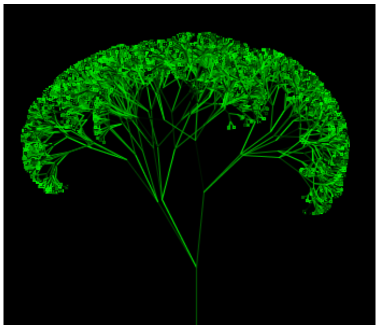
Write a function `void rotate(float theta, float a, float b)` which will rotate a point `(a,b)` around the point `(0,0)` by an angle of `theta` radians, using the formulae above, giving a new point `(newA,newB)`. This function could be used with `x` and `y`, or `x` and `z`, or `y` and `z`, so let's just call them `a` and `b` to make them generic.

Now we run into a problem with returning results from functions – only one value can be returned, and we need to return two (`newA` and `newB`). There is a way to solve this, but it's not covered in COMP 1010, so create two global variables `newA` and `newB` and put the answers there. It's not the "proper" way to do it, but it will work.

Now modify the `drawDotGrid` function to use the mouse to spin the cube. Before drawing each circle, use your `rotate` function to spin it by `mouseX/100.0` radians around the Y axis, and then `(height-mouseY)/100.0` radians around the X axis. (If this seems backwards, just think about it and visualize it – it's correct.) The spinning must be done *before* you add the "zoom" factor of 150 to the z co-ordinate.

You should now be able to "grab" the cube and spin it with the mouse.

Exercise 9. This is a hard one that uses concepts at the edge of functions that you officially cover in the next course. This uses **recursion** – functions that call themselves. Yep, that's right! Like those Russian dolls with one inside another, a function can call itself, which calls itself, which calls itself, and so on – but when does it stop?? That's the trick- the nested function calls need to stop at some point.



You'll make some code to generate a random tree like in the image to the right. The way you do this is with the following algorithm:

- Draw a branch. The first branch starts at the bottom of the screen and is straight up
  - Calculate the end point of the branch by using the starting point, the angle of the branch, and the length of the branch.
- From the end of the branch, you need to draw 3 new branches. Each branch should be a random shift from the existing branch angle. For example, one is slightly to the left, one is quite to the right, and one is straight up. Also, the new branch should be smaller than the old one, so they get smaller.
- Continue to draw three new branches at the end of each new branch, until the branch hits the minimum size. If it's the minimum size, draw a leaf (circle) at the end of it.

This is recursive. At level 1, you have 1 branch, the trunk. Then it splits in three, and you have three branch. Each new branch splits in three, so you have 3*3 branches. At each level, you multiply the number of branches by three. At the end, you have a lot of branches!

To let you focus on the functions and not everything else, I'll give you the globals from my solution:

```
final float MIN_BRANCH_SIZE = 5;
final float START_SIZE = 100;
final float START_ANGLE = 3*PI/2;
final float BRANCH_SHRINK_RATE = 0.7;
final float SPREAD = 1.7; // radians
final int BRANCHES = 3;
final float TREE_START_X = CANVAS_SIZE/2;
final float TREE_START_Y = CANVAS_SIZE-1;
```

The starting branch (the trunk) is `100` pixels. We stop building the tree when our branches get below the minimum size (`5`). Our first branch angle is `2PI/3`, which is straight up. Each time we generate a new branch, it is 70% the length of the prior branch. The new branches can spread within a `1.7` radians fan out from the existing angle. We make three new branches at the end of each branch.

The tricky part is turning this into a function. In this case, we only draw it once, so add `noLoop()` to the setup block so `draw` only runs once.

```
void draw()
{
  background(BG_CLR);
  drawBranch(TREE_START_X, TREE_START_Y,
             START_ANGLE, START_SIZE);
}
```

The logic for the `drawBranch` function, which takes four parameters: `x`, `y`, `angle`, and `length`, as above:

- If the length is too small
  - Draw an ellipse of size `length/2` at `x,y`, and return.
- Draw the line
  - Use the `x,y`, `length`, and `angle` to calculate the end point `X,Y` of the line
  - Draw the line
- Draw the branches
  - Use a `for` loop to go from `0<=i<BRANCHES`
    - For each branch, calculate the new angle `(random(SPREAD) – SPREAD/2)` and add to the existing angle.
    - Call `drawBranch` with the new `x,y` (the end point of the above line), the new angle, and a new length (old length multiplied by `BRANCH_SHRINK_RATE`).

That’s it! To make it green, I set the color to a random green variant for each line and leaf. This is a pretty short program.

Try playing with the parameters, the spread, shrink rate, etc., to see how the tree looks different.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 15. Compiling and the Java Virtual Machine

Summary

You will learn about the problem of computers only understanding binary, and how we get from our Processing program to running a program. You will learn about

- Compiling
- Virtual Machines

Learning Objectives

After finishing this unit, you will be able to …

- Describe the work flow that your computer takes, going from your Processing code, to a computer program running on your computer.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

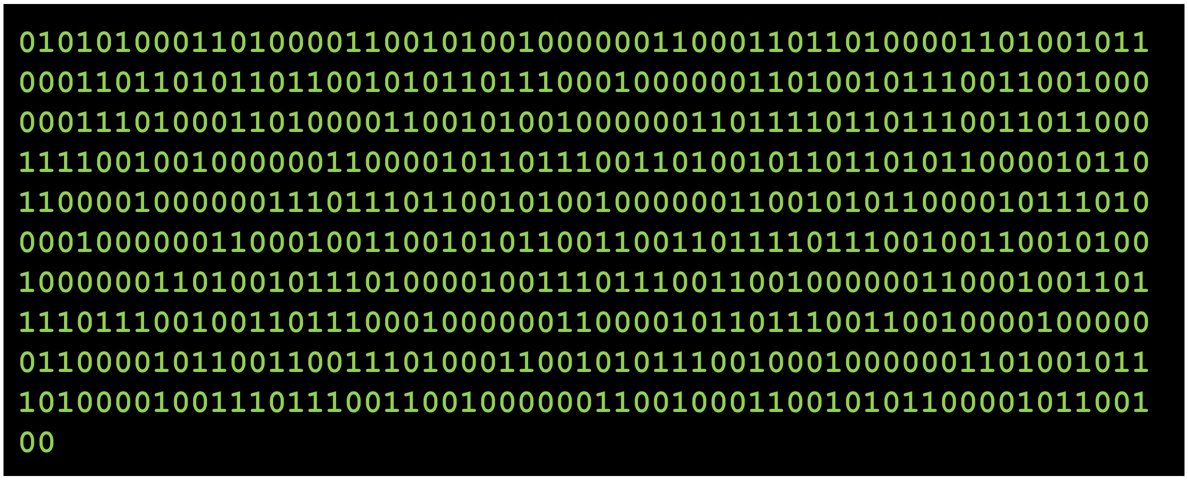
15.1 **Introduction**

In this unit we learn about a concept called compiling, and how this relates to the Java Virtual Machine. That's right – as we have learned, Processing is just Java, so it uses Java under the hood.

Although you do not do any programming in this chapter, this information is very important. Understanding what is happening under the hood of a computer is crucial foundational information for understanding computer programming. As you continue through your Computer Science career, you'll continue to fill out this knowledge. For now, let's learn a little more about the relationship between what you type in your Processing window, and, what the computer needs to be able to run your program.

15.2 **Compiling**

A fundamental problem facing computer programmers is that computers can only understand binary – a language of 1s and 0s. Here is a binary message, can you understand it?



```
01010100011010000110010100100000011000110110100001101001011
00011011010110110010101101110001000000110100101110011001000
00011101000110100001100101001000000110111101101110011011000
11110010010000000110000101101110011010010110110101100001010110
11000010000000111011101100101001000000110010101100001011101 0
```

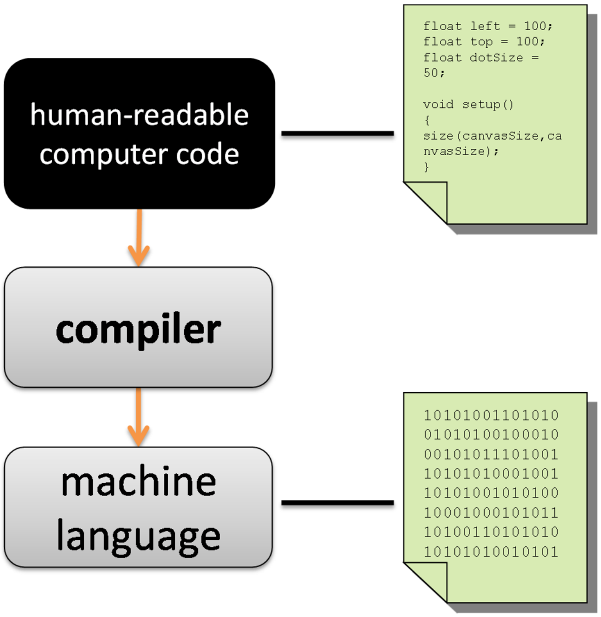
In this case, this binary code represents text, and is encoded as ASCII (see Unit 8). It gets even worse if we are trying to tell the computer to do things like light up a pixel or listen to the mouse.

Clearly, humans are not good at understanding binary. So, we invented computer programming languages, like Processing, which humans can use and understand more easily. Unfortunately, just as you cannot understand the binary above, the computer actually cannot understand the processing which is designed for people to read.

A special tool is needed to convert between the human-readable computer code, and the binary machine language, so that the computer can read, and execute, our program. Such a program is called a compiler: ***a compiler converts human-readable computer code into machine language***.

Without a compiler, the computer cannot understand what it should do.

Advanced: You may have heard of computer languages that do not need to be compiled, these are called interpreted languages. Instead of looking at a whole program, and compiling it in one go to machine language, interpreted programs generally convert one instruction or command at a time. This is very slow, but has some advantages; it can be more flexible, can potentially handle errors more gracefully, and often lets people write one-line computer programs to do small jobs.



```
float left = 100;
float top = 100;
float dotSize =
50;

void setup()
{
size(canvasSize,ca
nvasSize);
}
```

```
10101001101010
01010100100010
00101011101001
10101010001001
10101001010100
10001000101011
10100110101010
10101010010101
```

When you click on the run button (or press `CTRL-R`) in Processing, it automatically compiles and runs your program for you. If you go to basic Java, depending on the IDE, you will generally have to do this in two steps – compile first, and, if that was successful, you can then run it. In Processing, the compiled file is hidden from you and stored in a temporary system directory; you can't find it without a little digging.

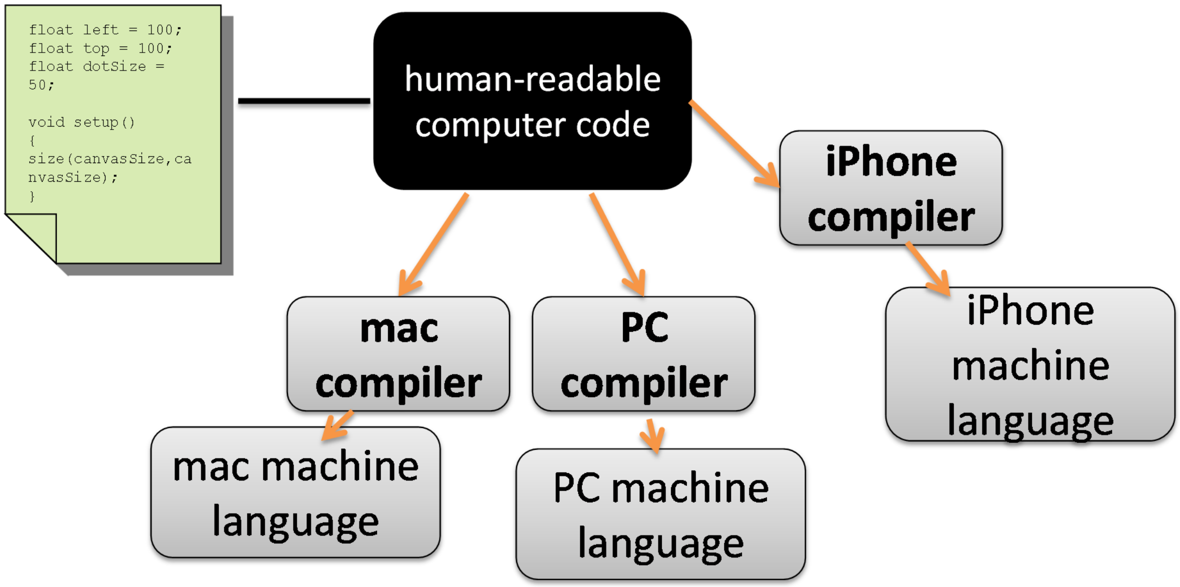
One nice thing about compiling is that, once it is done, you do not need to re-compile a program to run it again. You compile a program once, and then run it as often as you want. When you purchase or download software, it is generally already compiled, and you do not have access to the original human-readable source code.

15.3 **Machine Differences**

The idea of compiling seems pretty straight forward, but it ends up being highly complicated by the fact that different machines can speak dramatically different languages. That is, there is not one single machine language, but a whole range of them. Different chips, different generations, different manufacturers, and even sometimes different configurations of the same stuff, can result in differences in the required machine language. Different operating systems also are setup uniquely and so require different machine language to work well with them.

Clear examples of this in daily life are Windows-based and Mac-based machines, they speak different languages, or Android and Apple-based phones, which also speak different languages again. It gets even more complicated in the industrial and commercial world, as computer servers, embedded systems and controllers, custom hardware, etc., can all speak different languages.

The unfortunate solution to this problem is to have a wide range of compilers. Given one simple human-readable computer program, you need a whole range of compilers, one to target each machine you want your program to run on, since we need different versions of the machine language to get it to run on each machine. Consider the following graphic:



```
float left = 100;
float top = 100;
float dotSize =
50;

void setup()
{
size(canvasSize,ca
nvasSize);
}
```

To make things worse, each compiler and each language has its own quirks, so compiling your program for a new machine is almost never as simply as swapping out the compiler. Some things work differently, some features are missing, and so you need to do major changes to your program. For this reason, Mac and Windows version of the same software can be quite different, and may not even be released at the same time. This is such a big job that many companies only target one machine line.

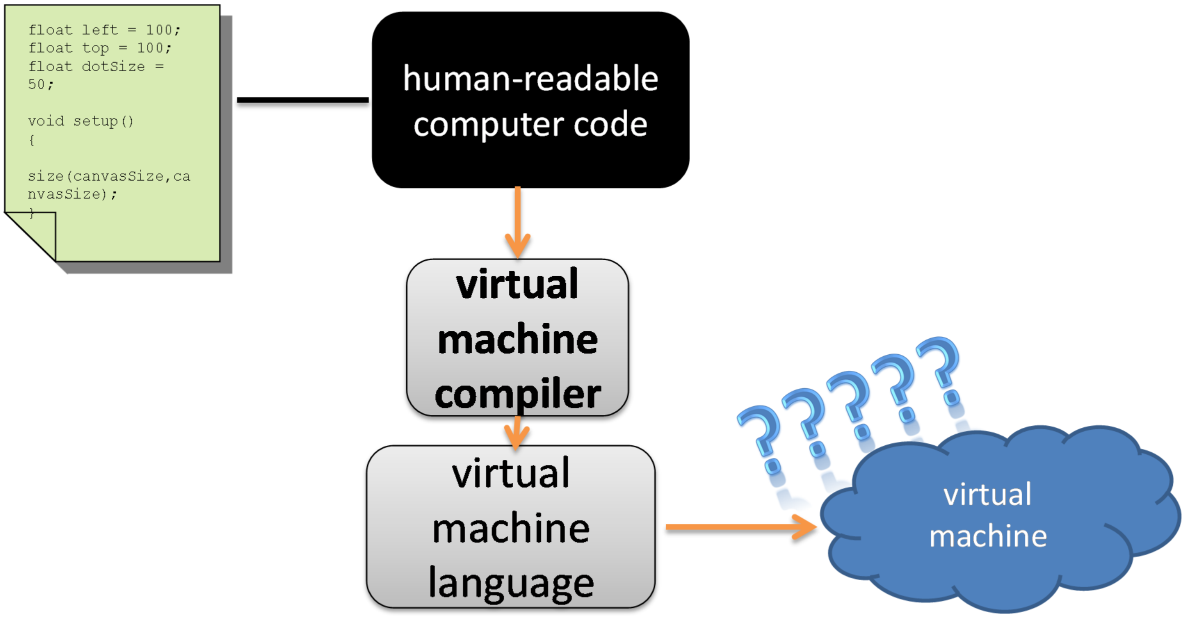
To make things worse, this situation does not scale well at all. If you have a new machine (say, a new version of iPhone, or a new competitor), you not only need a new compiler, but ***every single program you want to run on the individual machine needs to be painstakingly re-compiled and modified to target the new machine.*** This is a huge endeavor.

This ended up being a huge problem, particularly for business. Imagine you are a company with a very expensive web server running your software. One day, your server breaks, and you go shopping for a new one. Since the last purchase, the landscape has changed considerably, and you can now get a machine with 10x the computing power for half the price from a competitor – oh, but there's a downside, as the new machine speaks a different machine language. Your entire software library needs to be re-compiled to target this new machine, which greatly complicates the situation. There must be a better way!

15.4 **Virtual Machines**

There is a clever solution to the above problem, which takes a minute to wrap your head around. Instead of using a compiler to target a specific machine language, such as your PC or iPhone, Computer Scientists have developed a ***theoretical virtual machine: a machine with a very clearly-defined and well-documented machine language, and on that acts extremely predictably given some code to run. The downside? This virtual machine doesn't actually exist as a physical device, its fake***!

So, given some human-readable computer code, we use a virtual machine compiler to convert it to virtual machine language.



```
float left = 100;
float top = 100;
float dotSize =
50;

void setup()
{

size(canvasSize,ca
nvasSize);
}
```

This seems a little crazy – why go through all this work creating machine language for a virtual machine, which doesn't exist? Why would we do this?

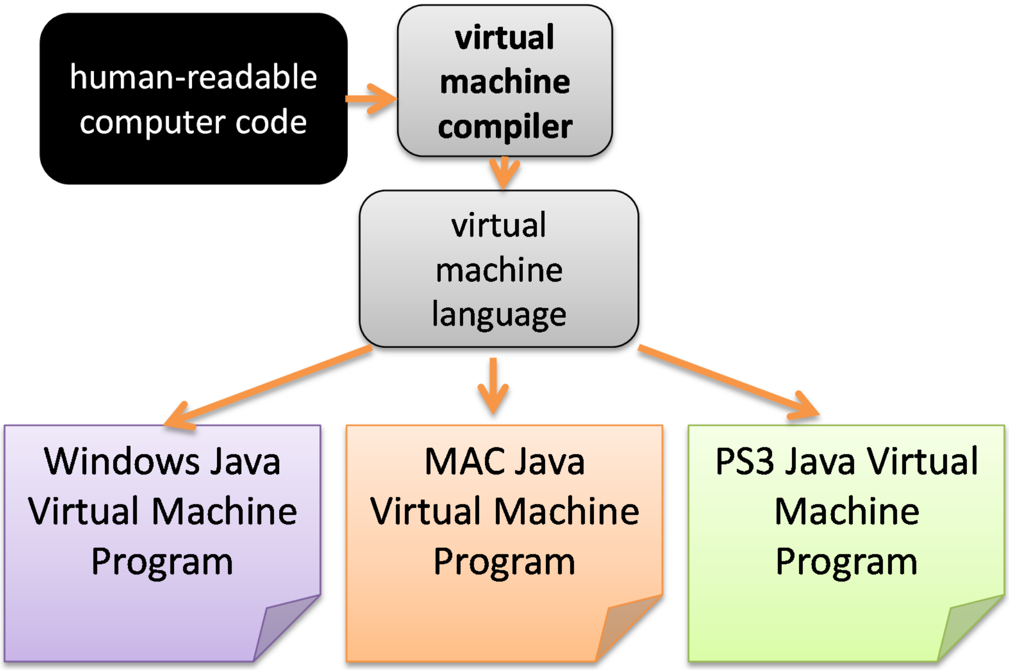
It turns out that, since the virtual machine is so well specified, we can create a program to emulate a virtual machine. We make a new program called a ***virtual machine that reads virtual machine language, and converts is on-the-fly to real machine language to run on our machine.*** If we do a good job at this, then our virtual machine can run any program that has been compiled to virtual machine language.

Java works like this. When you compile your program, it actually converts it first to Java's virtual machine language (which they call byte code), and saves that. Then, when you want to run your program, it starts up a Java Virtual Machine for your computer, and runs your program.

In Processing, when you click your run button, all of this happens invisibly to you. It compiles your program to Java byte code (virtual machine language), starts up a Java Virtual Machine on your computer, and tells that machine to run your byte code.

You may have noticed some time that you had to install a Java Virtual Machine (often packaged as a JRE, Java Runtime Environment, with a lot of other goodies) before to get programs to run. Luckily, Processing comes packaged with one installed with it, so you don't have to mess with it.

Java has virtual machine programs, or emulators, for a very broad range of platforms. If you have a Java program, all that you need to do to get it to run on a new machine is to ensure it has a Java Virtual Machine installed. In this class, you may have seen this first-hand, if you share a Processing program between Windows, Mac, or Linux machines. Processing even has Android and iOS versions.


human-readable computer code
virtual machine compiler
virtual machine language
Windows Java Virtual Machine Program
MAC Java Virtual Machine Program
PS3 Java Virtual Machine Program

This solution is extremely scalable. Going back to our earlier business example: if the business wrote their entire library in Java, changing to a new machine should be seamless, assuming that the new machine has a Java Virtual Machine available. When new hardware is introduced, such as a new type of cell phone, if someone writes a good Java Virtual Machine, then most existing Java programs should work seamlessly (assuming the machines have similar capabilities). There is no need to re-compile and tweak them! This is a good idea!

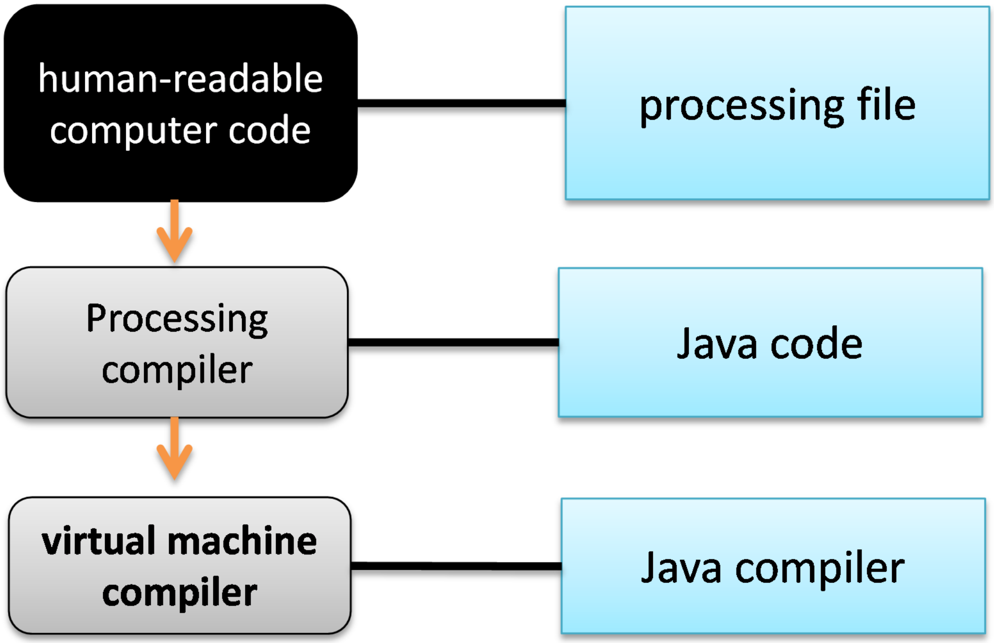
Virtual Machines do have some downsides. It can take a lot of extra time to start a program, as the computer needs to fire up a virtual machine, and toss your program to it. Virtual machines can also be slow, since it needs to convert from the pre-compiled virtual-machine code to the specific machine code. Finally, all of this can take more memory than traditional programs. Because of these reasons, performance-based software has been slow to adopt the virtual machine model. However, this is now changing. Very smart compilers and clever techniques such as Just-in-time compiling have made major improvements to the performance of Java programs, and so more and more performance computing is being done in such languages.

Finally, Java is not the only language to use this model, as the idea is growing. One popular framework that you may have heard of is Microsoft's .NET libraries, which works heavily on the same principle – however, Microsoft's focus has been more on very powerful functionality, where Java has been more about extreme cross-platform capability.

15.5 **Processing and Virtual Machines**

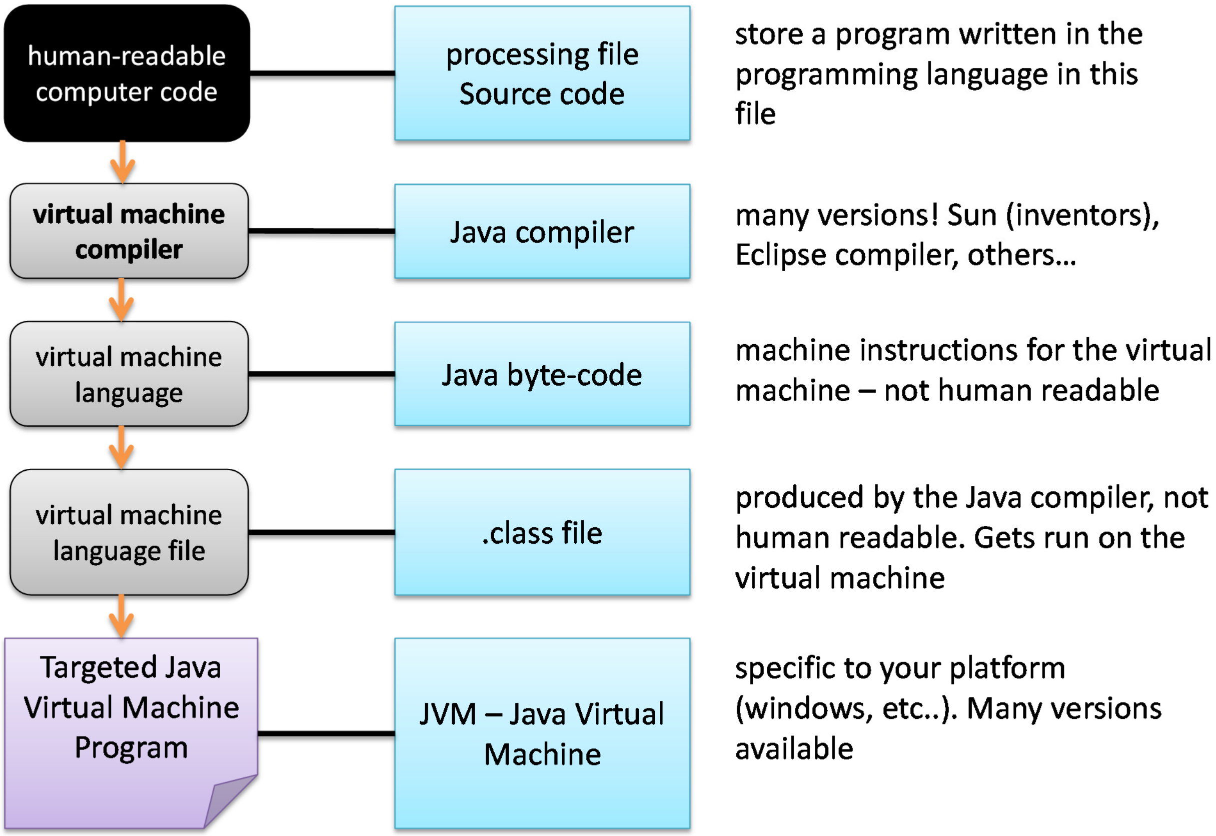
Most of this chapter has been about Java, not Processing – where does Processing fit in?

For Processing, there is just one extra step: Processing has its own compiler step that converts your Processing program to Java first. Then, all the above happens as normal:



15.6 **Overview and Terminology**

The final piece to cover is just to overview the entire process and to cover some of the common vocabulary. As shown in the following graphic, we start with our Processing file, often called the ***source code***. This gets converted to Java with the Processing compiler, which then gets converted to Java byte-code (virtual machine language) with the Java compiler. At this point, the computer stores this byte code in a special file called a `.class` file. If you have worked in Java, or poke around the secret Processing temporary folders, you can find this for your program! Then, this `.class` file is passed to a targeted virtual machine specific for your computer, and run.



human-readable computer code	processing file Source code	store a program written in the programming language in this file
virtual machine compiler	Java compiler	many versions! Sun (inventors), Eclipse compiler, others…
virtual machine language	Java byte-code	machine instructions for the virtual machine – not human readable
virtual machine language file	.class file	produced by the Java compiler, not human readable. Gets run on the virtual machine
Targeted Java Virtual Machine Program	JVM – Java Virtual Machine	specific to your platform (windows, etc..). Many versions available

Check your Understanding

15.7 Check Your Understanding: Exercises

Exercise 1. Using Wikipedia, look up the article for the Java Virtual Machine. This is often shortened to the JVM. We saw above the JRE. What is the JDK?

Exercise 2. Also using Wikipedia, look up Just In Time compilation (it has its own page).

a. What is it?
b. What are two benefits?
c. What are two drawbacks?

CuSn Exercise 3. People have really strong opinions about whether the Virtual Machine scheme used by Java is a good or bad thing. Do a google for "is java faster than `c++`", where `c++` is a fast, traditionally-compiled program, to see the kinds of discussions people are having on this. Who is right?

Ag Exercise 4. If compiling is the process of moving from human-readable code toward machine code, what is the reverse process? Can you take your favorite game, *de-compile* it, and modify it? Apart from the legal ramifications (this is illegal in some places), is this a good idea?

a. Find the Wikipedia page on *decompiler*. Give it a quick read – does this look like an easy topic? Also, read the legality section.
b. This falls under the larger umbrella of *reverse engineering*, which also has its own Wikipedia page. In addition to computer code, what other technologies are people reverse engineering?

Au Exercise 5. For the truly adventurous, you can look a little deeper at the compilation process in Processing. If you look inside the Processing main folder, you will see a program called `processing-java`. Open a command window here (the console in Windows, or terminal on a Mac or Linux), and run the program. Can you figure out how to use that program yourself to compile your Processing Sketch

a. Also notice that there is a `java` folder. Inside there, go into the `bin` folder (`bin` stands for binary, which means compiled programs). You will see a series of programs that help Java do its work, including `java` (Java Virtual Machine). If you become an expert Java programmer you will become very familiar with these tools.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 16. ARRAY BASICS

Summary

You will learn a new fundamental programming technique called arrays. Specifically, you will learn

- How to use arrays to overcome the limitations of creating individual variables
- How to make and use arrays in Processing
- How to use arrays with loops to quickly solve large problems

Learning Objectives

After finishing this unit, you will be able to …

- Create an array with any data type and size
- Store data in an array, and retrieve data from it
- Iterate through arrays with a loop
- Get the length of any array
- Initialize an array with literals

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

16.1 **Introduction**

Arrays are a fundamental programming technique used to address the limitations of creating individual variables. A great way to really understand what this means is to work through a quick motivating example.

Let's make a quick processing program to create a particle (small point) that fires off from the mouse pointer at a random speed in a random direction. If it hits the edge of the screen, start over from the mouse in a new direction at a new speed.

Our global variables include finals for `MAX_SPEED` (I set it to 5). In addition, the particle needs global variables to keep track of its current position, `ballX`, `ballY`, and variables to keep track of its speed in the `x` and `y` directions, `ballSpeedX`, and `ballSpeedY`.

In the setup, when the program starts, set the ball position to be the center of the screen, and generate random speeds for the X and the Y. **note:** make sure to include negative X and Y to go left or up. In my case, I created a function to help out:

```
float randomInRange(float min, float max)
{
  float range = max-min;
  float r = random(range);
  return r + min;
}
void setup()
{
  size(500, 500);
  ballX = width/2;
  ballY = height/2;
  ballSpeedX = randomInRange(-MAX_SPEED, MAX_SPEED);
  ballSpeedY = randomInRange(-MAX_SPEED, MAX_SPEED);
}
```

From here, the draw block is straight forward. Clear the background, move the ball (add the `speedX` and `speedY` to the position variables), and draw the ball. If it's off the edge of the screen, then start again at the mouse and generate a new random speed and direction. Here is my whole program:

```
final float MAX_SPEED = 5;
float ballX;
float ballY;
float ballSpeedX;
float ballSpeedY;
float randomInRange(float min, float max)
{
  float range = max-min;
  float r = random(range);
  return r + min;
}

void setup()
{
  size(500, 500);
  ballX = width/2;
  ballY = height/2;
  ballSpeedX = randomInRange(-MAX_SPEED, MAX_SPEED);
  ballSpeedY = randomInRange(-MAX_SPEED, MAX_SPEED);
}

void draw()
{
  background(0);
  if (ballY < 0 || ballY > height ||
      ballX < 0 || ballX > width)
  {
    // move the ball to the mouse
    ballX = mouseX;
    ballY = mouseY;

    ballSpeedX = randomInRange(-MAX_SPEED, MAX_SPEED);
    ballSpeedY = randomInRange(-MAX_SPEED, MAX_SPEED);
  }

  ballX += ballSpeedX;
  ballY += ballSpeedY;

  stroke(255);
  point(ballX, ballY);
}
```

This is interesting, but wouldn’t it look cool if we had more particles? Let’s try to update the program to three particles. To do this, we need to keep track of the positions and speeds of EACH particle. So, we need a bunch of new variables. I renamed my existing variables with a number for clarity:

```
float ballX1;
float ballY1;
float ballSpeedX1;
float ballSpeedY1;

float ballX2;
float ballY2;
float ballSpeedX2;
float ballSpeedY2;

float ballX3;
float ballY3;
float ballSpeedX3;
float ballSpeedY3;
```

Following, we need to repeat all the logic in the draw and setup blocks for the 2nd and third ball. We can cleverly use some functions to simplify this a little. However, there is only a limit to this, and it is very clunky. Try it on your own to really get a sense of this problem.

What if we wanted to have 100 particles? 1000? This quickly becomes crazy. Clearly there must be a better way – arrays! We will come back to this example later. For now, let's learn the fundamentals of arrays.

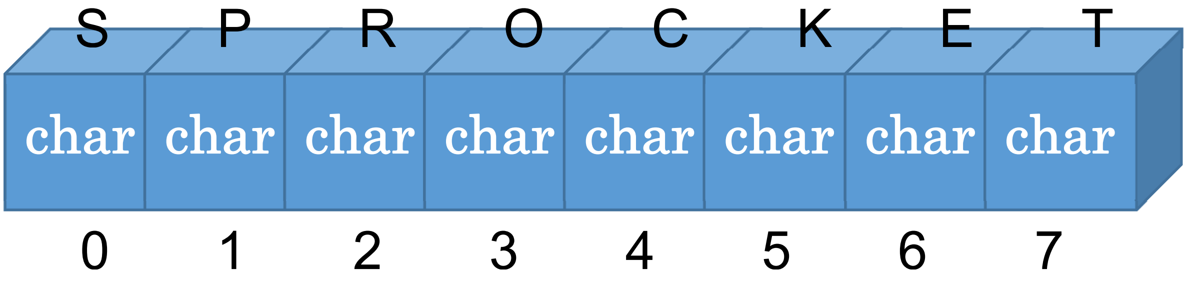
16.2 **What is an array**

An ***array is an ordered list of data of a given type.*** In English, that means a collection of a bunch of variables of the same type, where the collection is ordered. For example, you can have an array of 100 integers, an array of 10,000 strings, an array of 5 characters, and so on. In each case, you should imagine these collections as big lines of data, with one variable after another.

The power of arrays is that you can easily wrangle (create and use) large numbers of variables. Up until now, if you wanted 10,000 variables, you would need to type every one in and give every one a name. Using arrays, you can create 10,000 variables as easily as you can create 10!

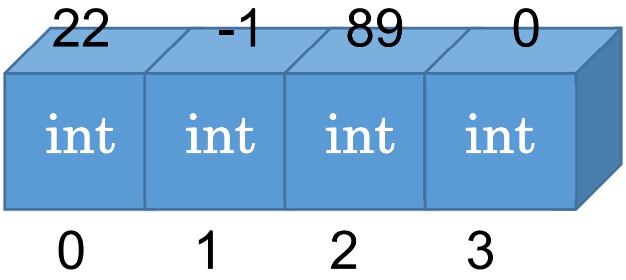
We have actually seen an array before. When we learned strings, we learned that, under the hood, it just is a bunch of characters. At the core, a string is an array of characters. As shown in the diagram, the String `"SPROCKET"` is actually made up of 8 characters lined up in a row. This is an **ordered list of data** as we talked about above.

When we learned this in the strings module, we actually already learned a lot about arrays. They have a given length. ***Each bin in the array has a designated number called an index***, and the numbering starts at 0. In this case, it is an array of characters of size 8.



Unlike strings, with general arrays we are not limited to characters, or, working with arrays through an interface like the `String` type. We can make arrays of any type, and, can work with them directly.

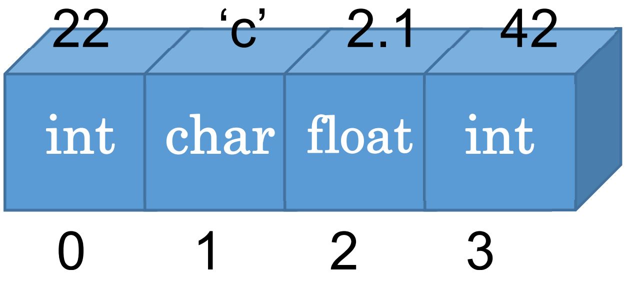
An array of four integers may look like the image here. As with strings, again, each box has its own unique number. The integers are all lined up. The first index is 0.



While arrays are great and quite flexible, there are some important key limitations:

Arrays are fixed length: Once an array is created, the length is fixed. When you create an array you decide how big it should be. This cannot change no matter what! BUT – what if you need an array to shrink or grow? This is a classic final exam question. There is only one way. Make a new array, and copy your data from the old one into the new one.

Arrays are homogenous: Every bin in an array is the same type. If you have an array of integers, every bin in the array stores an integer. You cannot have an array like in the inset. You have to choose the array type when you are creating it, and it cannot change.



Arrays are homogenous, ordered, lists of data of a single type. Let's learn how to make them.

16.3 **Creating Arrays**

Arrays work differently from anything else we learned in the course so far, and this comes up in many places. This is one of them. Unlike other variables, which only have one step, creating an array has two steps:

- ***Declaration:*** Declare the array variable container (as with other variables)
- ***Instantiation:*** Create the array (allocate memory) and assign it to the container.

Here is the syntax. To declare an array:

```
type[] variableName; // type is any Processing type.
```

You use the square brackets `[]` after a type to tell Processing that this is an array variable. For example:

Declare:

```
float[] studentScores;
String[] studentNames;
```

IMPORTANT. Array variables do not store the actual array (more on this below). It just remembers where in the computer memory the array is stored. As such, you do not specify the array size when you create the variable.

Once the array is declared, you need to instantiate it. This reserves the computer memory for all those bins you want.

Instantiate:

```
variableName = new type[arraySize];
```

`variableName` must already be declared, as in the previous step. You create a new array using the `new` keyword, and tell it the type and size. Once the new array is created, the variable stores where in computer memory the array is. At this point, this may seem like too much information, but this computer-memory approach helps to explain and understand a lot of array quirks.

Here is another example:

```
double[] studentScores;
studentScores = new double[30]; // 30 students in class
```

Of course, just like with other variables, you can combine the **declaration** and

instantiation into one command:

```
int[] studentAges = new int[30]; // declare and instantiate
```

Remember, once the array is instantiated, the size cannot be changed!!

Here are some more examples of valid array declarations and instantiations. Any valid Processing type can be used. Be sure that the type of the variable (**declaration**, on the left) matches the type of the array (**instantiation**, on the right), or it will not work.

```
double[] gameScores   = new double[5000];
String[] blogComments = new String[200];
long[] bigNumbers     = new long[100];
```

remember: you must **declare** and **instantiate** or you cannot use the array.

Although we haven't yet seen how to use arrays, now is a good time to highlight the power of the above examples. In each case, we are creating a large number of variables (5000, 200, and 100) without a lot of typing.

Finally, what is the maximum size of an array? This depends on the language. In Java, it's the maximum integer value, around 2 billion. In practical use, however, you won't use arrays this large. For bigger data sets you will use other techniques.

16.4 **Using Arrays**

So now we can create an array of any type and any size, but how can we actually use it in our code?

You can access any bin in the array just by using the array variable name, and putting the bin number in brackets:

```
arrayVariable[binNumber]
```

In this case, the result is that you have a variable that matches the array type.

For example:

```
int[] numbers = new int[10]; // array of 10 ints
numbers[0]  ← first int in the array
numbers[5]  ← sixth int in the array
numbers[9]  ← last int in the array
numbers[10]  ← ?? error
```

That last line will raise an error: Index out of bounds. The bin doesn't exist. This is the famous off-by-one error we have been seeing all course, so be careful!

Using the above syntax, you can now use the variable just like any other. For example, to store values in the array, you use the array variable and bin combination just like any other variable:

```
variableName[bin] = value
numbers[3] = 10;
numbers[0] = -1000;
```

Since this is an array of integers, you can store any valid `int` in the array bins.

Likewise, we can retrieve the values from the array bins just like any other variable:

```
int i = numbers[4];
if (numbers[1] < 100) { …
```

You can use this array variable – with the bin number in brackets – anywhere that a normal variable can be used.

When accessing the array bins, you can give any integer as the bin number. This could be a variable, a literal, even a calculation as long as it results in an integer. For example, this is valid:

```
int i = 5;
values[i] to access bin i
```

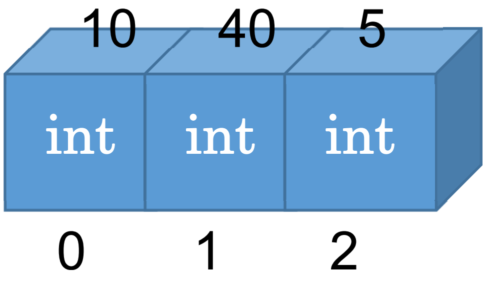
but it does not make sense to use non-integers:

```
values[5.5] ← doesn't make sense
values["yo!"] ← doesn't make sense
values[false] ← doesn't make sense
```

So now you have the tools to make large collections of variables, and to use them. It may not be obvious how all this fits together, so let's work through some examples.

16.5 **Example: first arrays**

Make an array of integers, size three, and put three numbers in it. Put the numbers 10, 40, and 5, so that the array looks like this:



Now, make an array of three strings and put three messages into it: "woah!", "hey there", "nice day".

Here is my code for this so far. First I create the variables and declare the arrays. Then, I store the data into the arrays.

```
final int COUNT = 3;
int[] sizes = new int[COUNT];
String[] messages = new String[COUNT];

sizes[0] = 10;
sizes[1] = 40;
sizes[2] = 5;

messages[0] = "woah!";
messages[1] = "hey there";
messages[2] = "nice day";
```

Use the integer array to set font sizes for the three messages, and draw it to screen.

Since the text is all different sizes, it's a little tricky to draw them to not overlap. My technique is to remember the line position (y), and before drawing the text, add the text size. This way, you move DOWN by the text size, and it draws UP from the position, into the space you just filled:

```
int y = height/2;
y += sizes[0];
textSize(sizes[0]);
text(messages[0], 0, y);

y += sizes[1];
textSize(sizes[1]);
text(messages[1], 0, y);

y += sizes[2];
textSize(sizes[2]);
text(messages[2], 0, y);
```

By using arrays, we were able to greatly simplify the variable creation process. Instead of copy-pasting to create three variables, we just created them once. However, we still copy-pasted the code to use the variables. If this example had 100 items, it would be a lot bulkier. Let's learn how to use for loops to simplify this.

16.6 **Iterating Through Arrays**

We can solve the above problem quite simply. Since the array index – the bin number – is an integer, we can use a variable there. Since we can use a variable, we can use a `for` loop to iterate through that variable! Piecing it all together, we can use a for loop to go through the bins to simplify the above code.

Notice how the code to actually draw the messages on the screen was identical, with the only change being the bin number used in the array? So, the above three drawing commands are equivalent to the following:

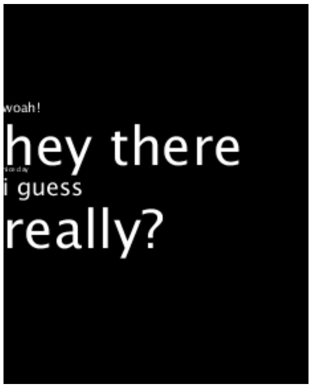
```
int y = height/2;
for (int i = 0; i < COUNT; i++)
{
  y += sizes[i];
  textSize(sizes[i]);
  text(messages[i], 0, y);
}
```

The `for` loop goes through 0, 1, 2, giving us the same output result as earlier.

The real power of arrays is in how nicely they scale up. In this case, let's add two more lines. To do this, all that we need to do is

- Make the arrays bigger
- Store more data in the arrays

Try it out yourself. The `for` loop does not need to change, only the arrays.



```
woah!
hey there
nice day
i guess
really?
```

16.7 **Example: random points**

Let's work through an example to illustrate the power of arrays. This program will generate 8 random points around the screen on startup. Then it will draw lines from those points to the mouse each frame. This will look like the points are stuck but the mouse can move. Finally, if the mouse button is pressed, generate a new set of random points.

First, let's create our arrays as globals. We should use a `final int` to determine the size of the array, and make an array for the `x` coordinates, and another array for the `y` coordinates. These need to be global so that they persist and do not get erased each time we draw:

```
final int POINTS = 8;
final int[] x = new int[POINTS];
final int[] y = new int[POINTS];
```

Now, we will need to generate random points on startup, AND, if the mouse is clicked. Instead of writing this code in two places, let's make a function to do it for us. This function should use a `for` loop to go through all the bins of the `x` and `y` arrays. Then, at each bin, store a random number. For `x`, we want this number to be within the range `0<=x<width`, and for `y`, we want `0<=y=height`.

```
void newPoints()
{
  for (int i = 0; i < POINTS; i++)
  {
    x[i] = (int)random(width);
    y[i] = (int)random(height);
  }
}
```

This code goes through all the bin numbers from `0` to `POINTS-1`, and creates a random `x` and `y` for those bins. Each bin gets a new random number.

Call this function in the startup code for the initial setup. And, in `draw`, use an `if` statement to check if the mouse is pressed, and if so, call this again.

Similarly, instead of putting all your drawing code into the draw block, make a new function called `drawLines`.

```
void drawLines()
{
  for (int i = 0; i < POINTS; i++)
   {
     line(mouseX, mouseY, x[i], y[i]);
   }
}
```

The code here goes through all bin numbers `0<=i<POINTS`, then draws a line from the mouse to that line.

Finally, the draw is simple. We clear the background, set the stroke, check the mouse pressed (and get new points if needed), and draw the lines:

```
void draw()
{
  background(0);
  stroke(255);
  if (mousePressed)
    newPoints();
  drawLines();
}
```

Run the program and you can see that we have 8 random points that the mouse draws lines to. Without arrays, imagine all the work this would have taken! We would have needed a lot more variables, and we wouldn't have been able to use the `for` loops for creating the random points and drawing the lines.

Now, scale the example up to 80 points. All that you need to do for this to happen, is to change the `POINTS` global, and everything else works!! Easy! 8 points is the same programming work as 80! Or 800!

16.8 **Example: mouse explosion**

Remember the example at the beginning of the unit? The one with a point shooting out from the mouse at a random speed and direction? Go back and type that up again – we're going to make it awesome.

So, let's see if we can modify the example to use arrays. First, let's make a global variable to tell us how many balls we need. For now, just set it to `10`. Next, *upgrade* those floats for the ball position and speeds to arrays, since we will need a unique position and speed per ball!

This is what I have:

```
final int BALLS = 10;
float[] ballX = new float[BALLS];
float[] ballY = new float[BALLS];
float[] ballSpeedX = new float[BALLS];
float[] ballSpeedY = new float[BALLS];
```

This now gives us 10 sets of variables, enough for 10 balls. Previously we only had one.

Now the program won't work, because we were using our variables, `ballX`, `ballY`, `ballSpeedX`, `ballSpeedY`, as floats, and now they are arrays.

The solution to this is that, each time we would do something to one of these variables, we need to upgrade to using arrays. Previously we would do something to one variable (e.g., add movement speed to position), but now we need to do the same work for *every* bin in the array. This means that we need to use `for` loops.

At the setup, we placed the original ball at the screen center. So, let's place all the balls there. Also, we set a random speed to the ball, so we need a new random speed for each ball as well. Wrap the existing startup code in a `for` loop, and update the variables to use the array, such that you do the same operation for all bins in the array:

```
for (int i = 0; i < BALLS; i++)
{
  ballX[i] = width/2;
  ballY[i] = height/2;
  ballSpeedX[i] = randomInRange(-MAX_SPEED, MAX_SPEED);
  ballSpeedY[i] = randomInRange(-MAX_SPEED, MAX_SPEED);
}
```

Similarly, in the draw block we need to repeat all the logic for *each* ball. We need to check each ball if it's outside the screen, and if so, move it to the mouse. We need to move each ball by adding the speed variables to its location. We need to draw each ball.

Again, wrap the existing code in a `for` loop to save a lot of work instead of treating each ball individually. Make sure to update each call to the old variable with the new array and bin number.

```
for (int i = 0; i < BALLS; i++)
  {
    if (ballY[i] < 0 || ballY[i] > height ||
      ballX[i] < 0 || ballX[i] > width)
    {
      // move the ball to the mouse
      ballX[i] = mouseX;
      ballY[i] = mouseY;

      ballSpeedX[i] = randomInRange(-MAX_SPEED, MAX_SPEED);
      ballSpeedY[i] = randomInRange(-MAX_SPEED, MAX_SPEED);
    }

    ballX[i] += ballSpeedX[i];
    ballY[i] += ballSpeedY[i];
    stroke(255);
    point(ballX[i], ballY[i]);
  }
```

Cool! It should work now. Try scaling up to 100, or even 1000 balls! To do this you only need to change one number.

16.9 **Array length**

When you create an array, you determine its length. That length is fixed and can never change. Also, knowing that length is useful, for example, when we use a `for` loop to go through an array.

So far, we use a named constant (a `final`) to set the size of the array, and use that same constant to determine how many bins are in the array. Sometimes, it's not so simple to keep track of how big an array is, or which global refers to which array. This is dangerous, because if you go off the end of an array – try to access a bin but the array is too small for that bin – your program crashes.

Processing provides a simple mechanism to get the length of an array. If we have an array variable, we can get the length with:

```
variableName.length
```

For example:

```
int numbers[] = new int[100];
println(numbers.length);
```

The output is 100, which is the number of bins in the array. As such, the last bin in an array is always `.length-1`. (remember! 0 based counting!).

This looks very similar to how we get the length of strings, but there is a very important difference.

For Strings:

```
String s = "Eric the Fruit Bat";
println(s.length());
```

And, for arrays

```
int[] numbers = new int[100];
println(numbers.length);
```

They look similar but there is a huge difference. Can you spot it?

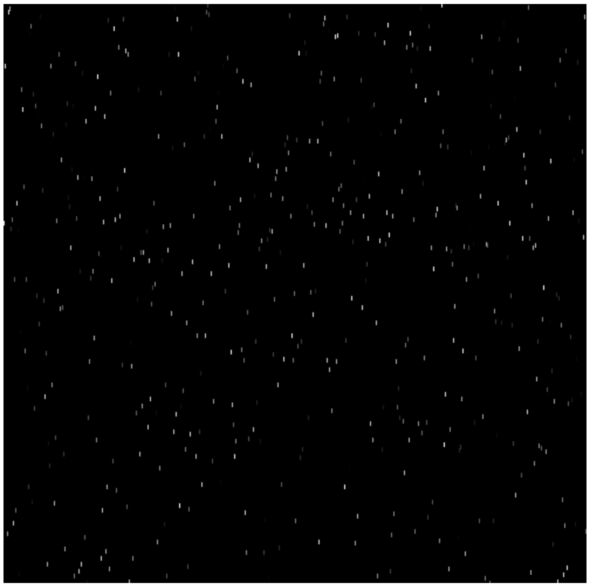
When you ask a string for its length, you need to put brackets at the end of the `length()`. For arrays, you don't use the brackets: `.length`. This is really annoying, and stems from the fact that strings are Objects and use object-oriented programming. Arrays are built directly into the language and are not objects. Unfortunately, we don't learn much more about this until a later course.

This difference is very confusing and probably makes little sense at this point. Unfortunately, you ***just need to memorize this. Arrays, .length. Strings, .length().***

Even when you know the named constant for the length of an array, you are safer using the array length. This is bullet proof, and even if something shifted or there is a mistake, `.length` always gives you the correct array size.

16.10 **Example: make it rain!!!**

Let's make it rain!



Make an array where each bin represents a rain drop. The bin number can be the `x` coordinate of the rain drop (so they are in every column), and we can store a random `y` value in the bin so that they are at random heights.

If we have more rain drops than `x` coordinates, use modulo to make it wrap around. For example, given a canvas size of `500`, bin `500` is a rain drop also in the left-most column, since `500%500` is `0`.

A quirk of this example is that we will generate a random number of dots in the startup. This means that we can't rely on a hard-coded final constant for the array size. It also means that we have to separate the array initialization and declaration.

We need to use the array throughout the program, and it cannot lose its data every time we draw, so the variable needs to be global. However, we don't initialize it to create a new array until the startup. Let's make the global variables. In this example we also use lines for rain (for artistic effect) so let's define the line length as well

```
final int DROP_Y_LEN = 10;
int drops[];
```

In the startup, let's instantiate our array and actually make some dots.

To generate a random number, let's be sure that we have at least one per column (width). Then, add a random amount, up to some maximum. I used a global for this and set the global to 10,000. Use this calculated number to create your array. ***NOTE: Make sure that you make it an*** `int`***, since you need to use an*** `int` ***to set the array size.***

```
int count = (int)random(MAX_ADDITIONAL_DROPS)+500;
drops = new int[count];
```

Now, use a for loop to set each drop's `y` value to a random position. This is a great place to try out the new `.length`. Be careful, you want a new random position for each dot. If you do this wrong, each dot will have the same `y` coordinate.

```
for (int i = 0; i < drops.length; i++)
{
  drops[i] = (int)random(height);
}
```

OK, we are all initialized.

The `draw` loop is actually pretty simple. We clear the background, and then use a `for` loop to go through each bin in the array. Since the size of the array was only stored in a local variable in startup, we no longer have access to that! We must rely on the `.length` property.

Calculate the `x` coordinate of the drop (`x` mod width – if it's bigger than width, just wrap around). The `y` coordinate is just the data in the bin. Use the drop line height to draw your rain drops:

```
for (int i = 0; i<drops.length; i++)
  {
    int x = i;
    int y = drops[i];
    stroke(random(256));
    line(x,y,x,y+DROP_Y_LEN);
}
```

All that is left is to animate them. A simple way is to simply add some value to the `y` coordinate (i.e., actually modify the bins of the array) each time. Use modulo to wrap around if it goes off the edge of the screen. Get this working before moving forward.

I think it's more fun, though, if the drops fall at different speeds. A simple trick we can do is to make there appear to be layers. What if the 0th, 3rd, 6th, 9th, etc., and so on moved at one speed, the 1st, 4th, 7th, 10th, etc., at another, and so on? We can do this with the following formula. Try to figure it out as an exercise

```
drops[i]=(drops[i]+i%3+1)%height;
```

Now that you're done, try to spice it up. What about color? Before each line, try this

```
stroke(random(256),0,random(256));
```

16.11 **Array Initialization with Literals**

Often times, as we have done, you want to pre-load an array with values. For example, we could use an array of strings to store the days of the week for the header of a calendar.

```
String[] days = new String[7];
days[0] = "S";
days[1] = "M";
days[2] = "T";
days[3] = "W";
days[4] = "R";
days[5] = "F";
days[6] = "S";
```

This is very useful, as now we can use this to draw the header of a calendar like we did in an earlier unit.

Go back and take a look (Section 13.2) at the code, type it up, we'll work on it. Of particular interest is the following ugly code:

```
// draw title bar
  int bottom = CAL_TOP+CAL_SPACE;
  int left = CAL_LEFT;
  text("S", left, bottom);
  left += CAL_SPACE;
  text("M", left, bottom);
  left += CAL_SPACE;
  text("T", left, bottom);
  left += CAL_SPACE;
  text("W", left, bottom);
  left += CAL_SPACE;
  text("R", left, bottom);
  left += CAL_SPACE;
  text("F", left, bottom);
  left += CAL_SPACE;
  text("S", left, bottom);
  left += CAL_SPACE;
```

We use a variable, left, to remember the leftmost spot for the letter. We draw a letter, then move the variable along.

Now that we have an array of days, we can replace all the above code with a simple for loop to get the equivalent output:

```
int bottom = CAL_TOP+CAL_SPACE;
int left = CAL_LEFT;
for (int i = 0; i < days.length; i++)
{
  text(days[i], left, bottom);
  left += CAL_SPACE;
}
```

Which is much nicer! However, we still have that bulky mess when we created the array – we had to set every bin individually.

When we create an array and want to store data into it right away, we can do what is called a literal initialization of an array. This is a shortcut that

- Creates a new array in memory. We don’t need the new command.
- Populates the array with the data we want. We don’t need to set every bin individually.

There is special syntax for this, and it looks like the following:

```
type[] variable = {element, element, element, …};
```

You just use the squiggly brackets `{` and `}`, and put your data in a list, separated by commas. In our above example, we can simply make our days array as follows:

```
String[] days = {"S", "M", "T", "W", "R", "F", "S"};
```

Processing automatically detects the length, allocates the memory for you, puts the data in the memory, and stores that memory address in days. All in one statement.

For example, what if I were to ask you to draw lines between the following sets of `x,y` points?

```
(100, 40) -> (140, 160) -> (40, 80) -> (160, 80) -> (60, 160)
-> first point
```

Previously you had to hard code this in a bunch of line statements. With arrays, maybe you could put them into arrays, and use a `for` loop to draw it. Previously, setting up the arrays would hardly be worth the effort. But with literal initialization, this gets much easier.

Let's make two arrays and initialize them with literals with the above numbers:

```
int[] xPoints = {100, 140, 40, 160, 60};
int[] yPoints = {40, 160, 80, 80, 160};
```

Now, we can draw them simply with a `for` loop. Let's go through all the bins, and draw a point from the current bin to the next one. We can use modulo to wrap around, so that when we go off the right edge of the array it goes back to zero:

```
for (int i = 0; i < xPoints.length; i++)
  {
    int next = (i+1)%xPoints.length;
    line(xPoints[i], yPoints[i], xPoints[next],
    yPoints[next]);
  }
```

What was the output?

Check your Understanding

16.12 Check Your Understanding: Exercises

Exercise 1. Create an array of floats with ten thousand bins, and practice both putting and retrieving data into and from the array.

a. Create the array with 10,000 bins.
b. Store the number 9999 in bins 0, 1, and the last bin.
c. Use `println` to print out the data in bins 0, 1, and the last bin

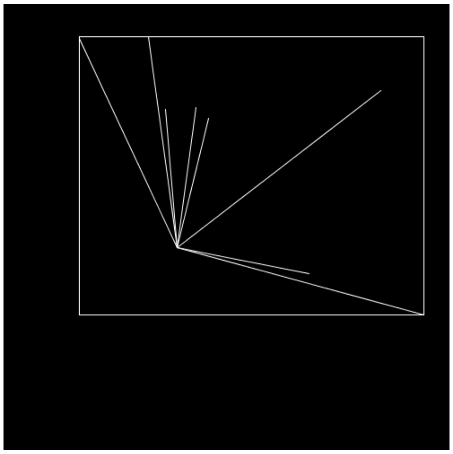
Exercise 2. Generate 500 random points (use two array variables, one for `x` and one for `y`), and draw them. Generate the points once, in the setup, and draw them each block.

a. Make the points move slowly to the right
b. Draw a line through the points, that is, from point 0 to 1, 1 to 2, etc.

Exercise 3. Make a program to plot the following points on a 500x500 canvas. Note: use literal array initialization and `for` loops.

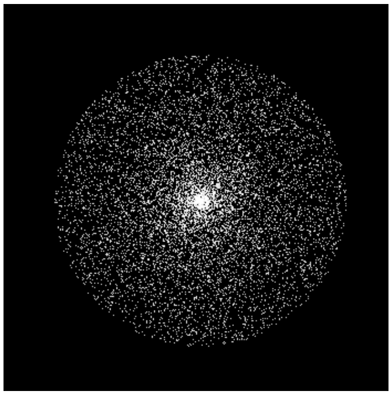
```
(150, 120), (158, 149), (166, 174), (175, 195), (183, 211),
(191, 224), (200, 233), (208, 240), (216, 245), (225, 247),
(233,  249), (241, 249), (250, 250), (258, 250), (266, 250),
(275, 252), (283, 254), (291, 259), (300, 266), (308, 275),
(316, 288), (325, 304), (333, 325), (341, 350), (350, 380)
```

Exercise 4. Update example 16.7 to draw a best-fit box around all the points, as shown here. This takes a little bit of work. The way to do this, is to find the smallest `x` and the smallest `y` in both the x and the y arrays, and then the largest `x` and `y`. Then, draw a box from the `smallestX,smallestY` to the `largestX,largestY`, and you have the box. How can you find the smallest and largest elements in an array? You need to do it manually, using `for` loops.



Exercise 5. Update example 16.8, the one with the points exploding from the mouse. Currently, the particles fire off randomly in any direction. Add gravity so that the balls fall toward the bottom of the screen, so that it looks like a fountain. Instead of thinking about gravity moving stuff (gravity doesn't move things, it accelerates them!), think about gravity accelerating things toward the ground. Therefore, each frame, you should add an amount of gravity to the ball's Y speed, not the position. Since gravity is `9.2m/s`^{2}, and we have `60` frames a second, you should add `9.2m/s`^{2} `* 1s/60f` = `0.153m/f`. Let's assume 1 pixel is 1 meter ☺, so you need to add 0.153 each frame to the speed.

In this exercise, you will animate the Big Bang at the start of the Universe! Draw a lot of "stars" (just single white pixels against a black background). Try 2500 stars to start with (but make that a named constant, of course). All the stars will start at exactly the same point (the center of the canvas – clearly the center of the universe), then they will "explode" outward, each with its



own random speed and direction.

For each star, you will need to know:

- Its current location (both X and Y)
- Its speed and direction of motion, which should be stored as separate X and Y motions (the change in X and change in Y to use each frame).

Since you have many stars, you will need to use arrays, and since you need to remember the location between frames, use global variables.

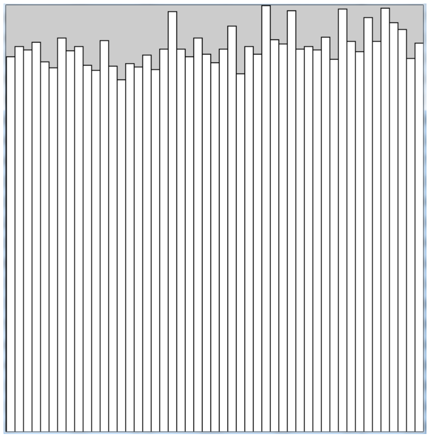
In the setup function, initialize all of the elements in all four arrays. The positions are easy: every star should start at the center of the canvas. For the velocity of each star, generate a random angle `theta` (between `0` and `2*PI`), and a random speed (from `0` to some maximum speed – try `3.0`, and of course use a named constant for this). Split the speed into separate `x` and `y` velocities by multiplying the speed by `sin(theta)` and `cos(theta)`, in the usual way. Note: It's a mistake to try to avoid `sin`/`cos` by generating separate random numbers for the `x` and `y` velocities. Do you see why? Try it! You get a rather strange universe that way.

In `draw`, draw each star as a point, and then move it to the next position by adding its velocities to its position. For an even better effect, give each star a different color. Add some randomness each time you draw a star so that they'll "twinkle".

Boom!

a. For the ultimate effect, use the perspective technique from the exercise in unit 14 to make the explosion 3D! Generate `(x,y,z)` locations, use velocities in a random 3D direction, and apply perspective.

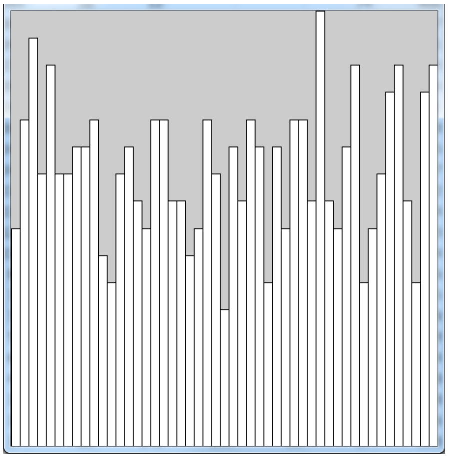
Exercise 6. Update example 16.10 to make the rain go in an angle. Make it both draw the line in an angle, and, move in an angle. This is a little tricky, but the array component should be easy.



Exercise 7. In the real-life Lotto 6/49, as of Nov. 16, 2015, a total of 19,920 numbers between 1 and 49 have been generated, in 3,320 draws held since 1982(**) (not including the "bonus" numbers). The top histogram(*) (bar graph) on the right shows how many times each of the 49 numbers have been drawn. The number 28 was drawn only 378 times, but the number 31 was drawn 450 times. Does this mean that the game isn't random? That 28 is unlucky? That 31 is lucky? Let's run a simulation and compare it to

real life.

The program will generate NUM_DATA_ITEMS (say, 500) random numbers between 1 and MAX_NUMBER (49). Then it will count how many times each possible number was generated (the number's frequency). Finally, it will display a very simple histogram showing the results, as shown at right. The bottom histogram shows one possible result when only 500 numbers were generated. It looks a lot more "random" doesn't it? You'll get a different result every time.



Complete a void generateData() function. It should create an array of NUM_DATA_ITEMS integers, each from 1 to MAX_NUMBER (note: 1 not 0!) and store it in the global variable theData.

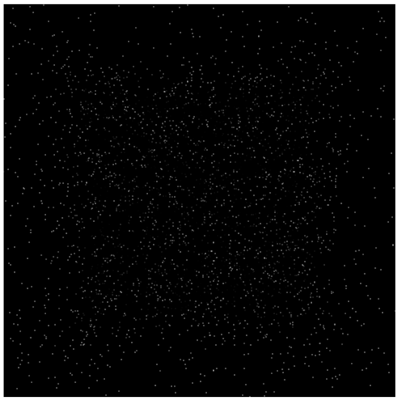
Complete a void findFrequency() function. It should create an array of MAX_NUMBER integers and store it in the global variable frequency. It should count the number of times that each integer x occurs in theData, and store that count in frequency[x-1]. (The -1 is because the lowest number is 1, not 0.)

Finally, complete a function plotHistogram() which will draw a histogram of the data in the frequency array. It will first need to scan through that array and find the maximum value that appears. The bar for that element should be the full height of the canvas, and all other numbers should be scaled accordingly. The full width of the canvas should be divided into MAX_NUMBER bars. The bottom of all of them should be at the bottom of the canvas.

Just for fun: Change NUM_DATA_ITEMS to 19920 and try it again. How do your results compare to the real Lotto 6/49 results? Are the real Lotto 6/49 numbers similar to your "random" ones? Do you think the game is really random?

(*) For a discussion of Histograms, see the Wikipedia page at en.wikipedia.org/wiki/Histogram

(**) The Lotto 6/49 data comes from www.lotto649stats.com/position_frequency.html



Exercise 8. This question will use the perspective technique introduced in Unit 14 exercises, using perspective to simulate a 3D image. This time, you'll draw a field of "stars" (single pixels), and make them fly toward you, or away from you, by changing their z coordinate. (Remember, the z coordinate controls

how far "away" or "into the screen" an object is.) The mouse will be the control for your spaceship – when `mouseX` is in the center of the canvas, the spaceship will be stopped, and the stars won't move. Moving the mouse to the right will make the stars move toward you (as if you were flying forward through space), and moving the mouse to the left will make the stars move away from you.

Use three `float[]` array variables, `starX`, `starY`, and `starZ` to hold the positions in 3D space of `STAR_COUNT` stars. The star with index `i` will have a virtual 3D position in space of `(starX[i], starY[i], starZ[i])`.

Write a function `void generateStar(int i)` which will create a random star and store its 3D location in the global variables `starX[i]`, `starY[i]`, `starZ[i]`. The stars should start with `x` and `y` coordinates from `-width/2` to `+width/2` and `-height/2` to `+height/2`, respectively. (These are virtual coordinates, not canvas coordinates, and the virtual `(x,y)` point `(0,0)` – the "vanishing point" – will be in the center of the canvas. That's where the stars should come from, or go to.) The `z` coordinate should be between `0` and `MAX_Z` – a predefined constant giving the maximum possible `z` coordinate for visible stars.

Complete the setup function by creating the three arrays needed, and filling them with random star positions, using the `generateStar` function.

Create a `void drawProjectedPixel(float x, float y, float z)` function (modify from the Unit 14 example) to project and draw the point. As we want to make stars with large `z` coordinates look far away, use the `z` coordinate to control the color ("brightness") of the star. Stars at `z=0` should be white, and stars at `MAX_Z` should be black, with all others at an intermediate shade of gray.

Complete `draw` to animate moving through a field of stars. Each frame, use `drawProjectedPixel` to draw all of the stars against a black background. Move every star by changing its `z` coordinate (`x` and `y` never change). All stars should change `z` by the same value from `-MAX_SPEED` (when the mouse is on the far right) to `+MAX_SPEED` (when the mouse is on the far left). Use an `int` not a float for the speed, so that it will be easier to make it `0` and stop the spaceship. Finally, regenerate the stars so that you never run out of them. When any star gets too close to draw `(z≤0)` or too far away to see `(z>MAX_Z)`, use `generateStar` to create a new one to take its place.

a. Yes, a double gold exercise! Modify the star field so that you can rotate it in an arbitrary fashion, using the technique shown in Unit 14 exercises. The challenge here is how and when to apply the rotation. Hint: the star coordinates themselves should NEVER change, only when you project do you rotate.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

UNIT 17. ARRAYS AND MEMORY

Summary

You will learn a about how arrays are stored in computer memory, specifically:

- The difference between data in a variable and data in general memory
- Basics of how a memory address is stored, and how to say “no address”
- How this impacts basic array operations like copying and comparing

Learning Objectives

After finishing this unit, you will be able to …

- Determine the default value stored in your arrays immediately after creation
- Use and compare variables against `null`
- Copy an array
- Compare an array

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

17.1 **Introduction**

Unlike other variables we have used so far, to use an array, you cannot simply make a variable and then use it. We have a new step, we need to instantiate the array with the `new` keyword.

Why this new step? Why don't we do this with our other variables? Well, we don't do it with any of the primitives at all, because what is stored inside the variable is actually the data. This may sound obvious, but in the case of arrays, the ***array data isn't actually stored in the variable, it's stored somewhere else***.

This is also the case for strings, and any objects once you learn object oriented programming. In Processing, a lot of the mess of Strings is hidden from you with nice syntax, but weird things are going on behind the scenes.

When you create an array variable, just like any variable, you need computer memory to store the data. For arrays, the computer doesn't know how much memory you need until you instantiate the array, because it doesn't know how many bins you will need. Therefore, it doesn't know how large to make the variable. The solution is the following:

- We store arrays in general computer memory (not in the variable), so we can decide how large it needs to be later.
- ***The array variable only stores the address of where the array is in general memory.*** We need to store this address to keep track of where the array is, after we create it
- We know how large an address is, so Processing is happy to create your variable.

To reiterate:

The array variable stores an address to the array. Until we give it an address, the variable is uninitialized, just like any other variable.

```
int[] intArray; // variable stores an address to an array.
```

When we know how large we want our array, we need to ask the computer for some memory.

The `new int[100]` command actually is asking the computer to go off and find us some empty memory, large enough to store 100 integers (how many bytes is this?). Once the computer finds this memory, reserves it (puts up a fence, marks it as taken), and gives us the address. Imagine the computer is going land shopping for you, finds just the right size, puts up tape so no one else can take it, and tells you where it is.

Processing stores that address in our variable.

```
intArray = new int[100];
```

To reiterate, the `new` keyword goes off and finds a chunk of memory big enough to store `100` integers in this case, and gives the address, which is then stored in `intArray`. Most of this time, this memory issue is invisible to you. When you use arrays, you don't usually think about memory addresses.

```
intArray[5] = 4;
```

However, the above command actually takes a trip out to the memory that you have saved, takes your 4, and stores it in the 5th bin in your memory.

We can actually take a peek at the memory address, just to prove a point. This is not only advanced, but not actually very smart – only really advanced programmers, and even then, only in very specific situations, look at memory addresses. For most of us, it's a waste of time. However, for the purpose of peeking at what is happening, here is a silly trick to get Processing to give us the memory address. Concatenate the array with a string.

```
println(""+intArray);
```

The memory address will be something weird, like `[I@1f3f158`. Computers have a very systematic way of storing memory, which doesn't really make sense to us.

As an example, when we create our variable, we just get a variable in our program.

```
int[] intArray;
```

However, when we create a new array:

```
intArray = new int[100];
```

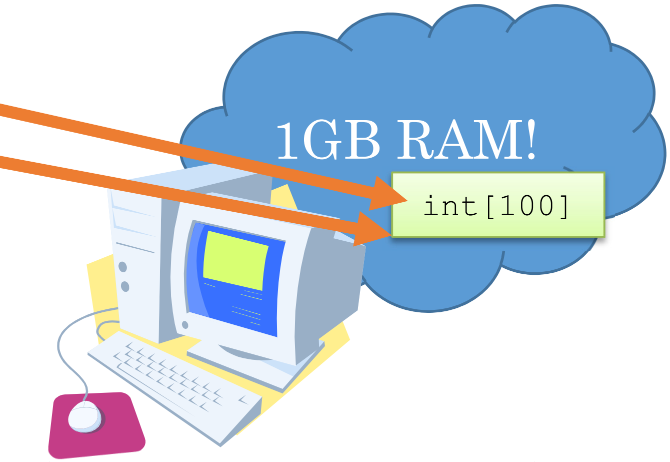
The computer searches its memory, finds enough space for our 100 integers, and marks off the space. Every time we use `intArray`, it actually goes off to that address and works there. E.g.,

```
intArray[4] = 20;
```

Goes off to that plot of memory, puts a number 20 in bin 4, and comes back.

Here is a series of visuals of what is going on. Here is our modern computer with loads of ram.

```
int[] intArray;
intArray = new int[100];
intArray[4] = 20;
```



(modern looking computer, Word clipart)

When the intArray variable is created, it has no memory address, and cannot be used. After new is called, then intArray stores the address of the section of memory we have to store our array data. Then, when intArray is used, the computer actually goes off and changes that memory.

Although I said we generally don't think about arrays and memory, there are some key times when we do, so you need to clearly understand this. Particularly when you start tossing arrays to functions, and wanting to do things like shrink or grow an array, you need these fundamentals down pat.

17.2 **Default value of array entries**

In Processing, what happens when we create a variable, do not initialize it, and then try to use it? For example:

```
int variable;
println(variable);
```

If you try to run this, you get an error. Processing says "the local variable variable may not have been initialized." This makes sense. Since we put nothing in the variable, what should the println do?

This is a feature of many languages. The compiler looks and can detect this instance, and refuse to use a variable that has never been set to a value.

Unfortunately, this isn't a trivial thing for the computer check. When working with arrays, it gets even harder. In fact, Processing just gives up. Let's try the following similar example. We create an array variable, instantiate the array, but do not put any data in the array bins:

```
int[] intArray = new int[10];
println(int[0]);
```

This time it works! Huh? Why? That variable is not initialized!

Because it's hard for the computer to detect which array bins are initialized and which are not, Processing has a workaround. When you create an array, it does a bit of work and fills the array with default values. This way, we know what is in the variable. If you imagine the land analogy I used earlier, when the `new` keyword finds and cordons off your new memory, it sends in a bulldozer to clear the memory to some default state.

What are these defaults? The defaults depend on the data type.

- For integer types, including the character, the default value is `0`. Careful (Advanced): for characters, this is the ASCII code `0` and not the character `'0'`.
- For Booleans, the default is `false`
- For Strings (and all objects), the default is `null`

Wait – what? `null`? What the heck is `null`? We'll get to that very shortly.

Generally be careful of relying on default values. That is, you may think: I know that my numbers get set to 0, so I will rely on that fact. This is not a bad idea, but the problem is, other languages may treat this differently, (e.g., C or C++) creating a very hard to find bug! Be clearly aware of the parameters of your computer programming language.

Advanced: Depending on the compiler and run-time environment, C or C++ programs may not clear the array memory at all, and give you whatever data was there previously. That's right, who knows what could be there. Make sure you understand the default value policy of the language and platform you are using.

17.3 **null – no memory address**

Remember how we talked about how array variables store the memory address to the array? Strings (and all objects) actually work the same way. That is, your string variable only stores the address of where the string is in memory. Again, nice Processing syntax hides this from you, but the string itself is stored off in memory the same as an array.

What if the variable does not yet have an address in it? ***Computers have a special value which means no address. In computer science, this is called null: no memory address.***

For example:

```
String[] names = new String[10];
println(names[0]);
```

your output is `null`. Try it.

Processing has a special keyword, `null`, that you can use to refer to this special value. For example, you can check if an array bin in a string array is empty by comparing it to `null`:

```
String[] names = new String[10];
if (names[0] == null)
  println("bin 0 is empty");
```

But! You may be thinking that we should never use == with strings. We'll come back to this later in this unit.

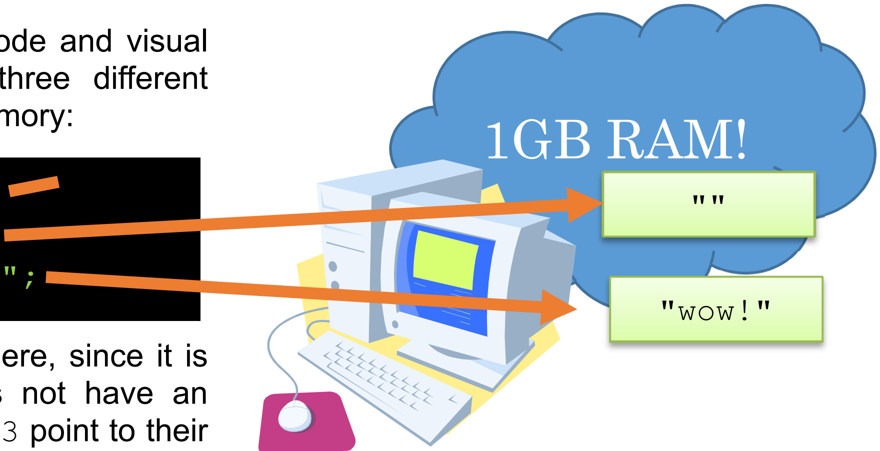
Another point: ***null is not the same as the empty string!*** This is confusing as heck, but an empty string is still a string, with length `0`. `null` is not a string at all, and if you try to check its length, the program crashes.

```
String[] names = new String[10];
println(names[0].length());
```

You get a problem called "null pointer exception." Means there is no string to check the length of.

Look at the following code and visual that shows how the three different strings are stored in memory:

```
String s1 = null;
String s2 = "";
String s3 = "wow!";
```



(modern looking computer, Word clipart)

`s1` doesn't point anywhere, since it is set to `null` and does not have an address. Both `s2` and `s3` point to their own respective strings in memory. The empty string is still a string, the same as "wow!", it just has length 0.

Finally, null cannot be used for primitives. You cannot set an integer or float variable to `null`, since these variables store your actual data and not a memory address.

17.4 **Copying Arrays**

How do you make a copy of a primitive variable like an integer or floating point? This should be obvious:

```
int i = 1982;
int j = i; // copy i into j, j now 1982
i = 1999; // replace i
```

```
println(i + " " + j); // what is the output?
```

The output is, as you'd expect:

```
1999 1982
```

`i` has the new value, 1999. `j` remembers the copy it took from `i` earlier, 1982. This works as you should expect, no surprises here. Let's do something similar with an array

```
int[] i = {1, 2, 3};
int[] j = i; // copy i into j..?
i[0] = 1999;  // change i only
println(i);
println(j);
```

What is the output?

```
[0] 1999
[1] 2
[2] 3
[0] 1999
[1] 2
[2] 3
```

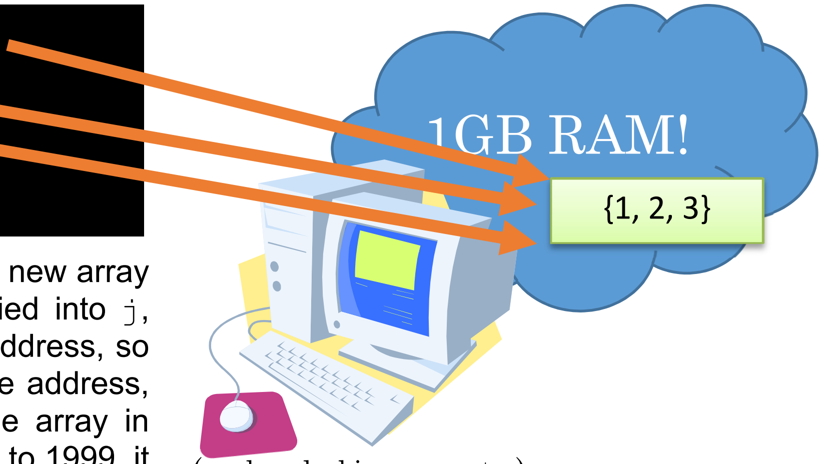
The top array is `i`, and the bottom is `j`. `i` originally had `1` in bin `0`. We copied `i` to `j`, hopefully preserving that old value, so we expect that `j` is the old `i`. Then we modify the `i` array to store `1999` in the bin. However, **both** arrays are updated!! This is confusing, how can we update two arrays just with one command?

In order to understand what happened we need to think again about arrays and memory. First, let's create `i`. We have a variable, and create a new array off in memory to store the `1,2,3`. The variable points to the memory address.

When we create the array variable for `j`, it doesn't point anywhere at first. When we set it to `i`, Processing does exactly what it does for all variables: copy the values! Unfortunately for us `i` does not hold the array, but rather, holds the memory address of the array. Therefore, Processing copies the memory address from `i` to `j`, and `j` gets the same memory address.

Consider the following graphic and code:

```
int[] i = {1, 2, 3};
int[] j = i;
i[0] = 1999;
println(i);
println(j);
```



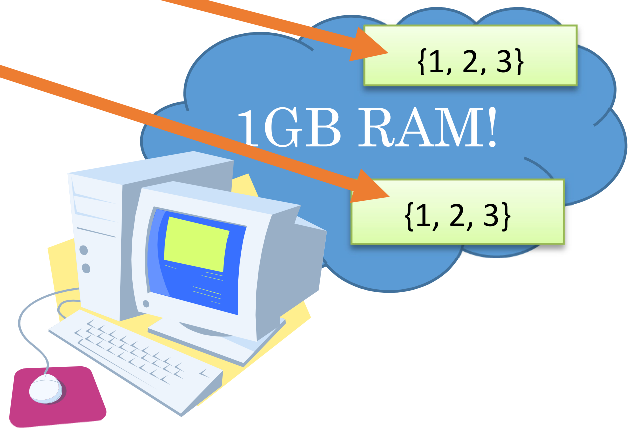
(modern looking computer)

When `i` is created, we get a new array in memory. When `i` is copied into `j`, however, we just copy the address, so both `i` and `j` have the same address, they both point to the same array in memory. When we set `i[0]` to 1999, it gets the address from `i`, goes off to memory, and changes that single array. As such, both the `i` and `j` arrays appear to be changed.

This didn't copy the array at all, just copied the memory address from one variable to another. We now have two variables that work on the same piece of memory, the same array.

What we wanted was this: (not working code)

```
int[] i = {1, 2, 3};
int[] j = ..? somehow copy i, and save into j..?
```



(modern looking computer)

We want an *actual* copy in memory, for Processing to create a new array, so that we can modify one and the other doesn't change.

In this visual, if we change the first bin in the array pointed to by `j`, then this doesn't impact `i`. Our code above doesn't work, but how can we achieve this?

This requires two steps:

- Create a new array with the same size.
- Copy the data over from the old array to the new one.

That is, we have to do it manually.

We can create a new array using the syntax we already know. Then, we can copy data over from the old array into the new one using a `for` loop.

```
int[] i = {1, 2, 3};
int[] j = new int[i.length]; // new array same size as i

// copy i into j
for (int bin = 0 ; bin < i.length; bin++)
  j[bin] = i[bin]; // bin-by-bin copy

i[0] = 1999; // only i is modified. j is different array
println(i);
println(j);
```

17.5 **Comparing Arrays**

Comparing arrays introduces similar problems as copying arrays. Our traditional mechanisms do not make intuitive sense unless we think about arrays and memory. Look at the following example:

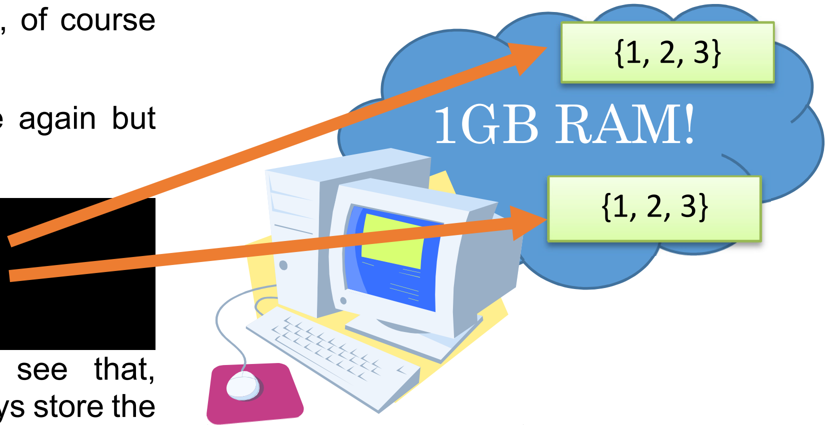
```
int[] i = {1,2,3};
int[] j= {1,2,3};
println(i==j);
```

What do you expect will happen? We can see that the arrays have the same data stored in them, so if we use `==` to compare them, we would expect to get `true`. However, if we run this, the result is `false`.

The reason for this is that the ***== compares what is in the variables: the memory addresses.*** Since we have two arrays, they are at different locations in memory. When we compare them, of course they are false.

Consider the same code again but with the visual aid:

```
int[] i = {1,2,3};
int[] j= {1,2,3};
println(i==j);
```



(modern looking computer)

Here, you can clearly see that, although the `i` and `j` arrays store the same data, they have different addresses. Therefore, the result of the == operation is false.

In fact, ***this is why you cannot use == for strings***. It just compares the memory addresses of where the strings are stored, which is generally not what we want. If we want to see if two strings are equal to each other, we actually need to compare string down to the characters themselves.

So, when we compare arrays, what we really want is a deep comparison: we need to look at each bin individually and see if they are equal across the arrays. We can do this with a `for` loop, and the following algorithm:

- Assume the arrays `a,b` are equal
- Go through each bin `i` in the array with a `for` loop
    - if `a[i]` does not equal `b[i]`, then the array isn't equal

In code:

```
// assume we have two int arrays, a and b, of the same length
boolean equals = true;
for (int i = 0; i < a.length; i++)
{
  if (a[i] != b[i])
    equals = false;
}
```

At the end of this, if there is any bin `i` where the arrays do not match, then the arrays are considered to be not equal.

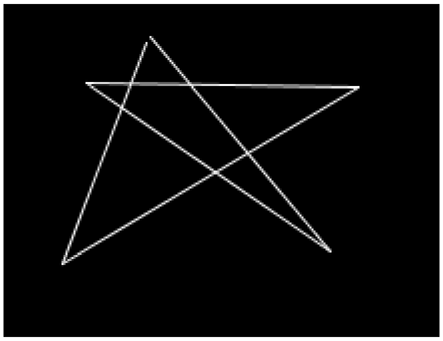
What if the array is very big, and, we find out very early (in the first bin!!) that the arrays are not equal? The above code will keep on trucking and continue checking the entire array – a complete waste of work. How can we quit the `for` loop as soon as we find out they are not equal? We can modify the boolean test in the loop to continue also while they are still considered to be equal:

```
// assume we have two int arrays, a and b, of the same length
boolean equals = true;
for (int i = 0; i < a.length && equals; i++)
{
  if (a[i] != b[i])
    equals = false;
}
```

The `for` loop will quit when either of the conditions are false: not equals, or, `i` is not less than length.

With strings, we do not need to do this work because someone has already written the code for us. We get to use that function using the method `.equals` on the strings involved. However, under the hood, they do a very similar operation.

17.6 **Example: wandering star!**



Let's make a program where we have a 5 pointed star, with its points wandering around (moving randomly). If the mouse is clicked, return them to their original locations.

We will do this by making two arrays for the points of the star, one for `x` and one for `y`. In the draw, we use a `for` loop to draw lines between all the points, and, move each point randomly. When the mouse is clicked, we want to return to the original points – but how do we do this? As we modify the points to move randomly, then we lose the original positions! We need to copy them, to back them up.

We need two sets of arrays

- The original points
- A copy of the points, that we modify to make wander around. We can change this copy without changing the original.

First let's make the original points. I will give you the data:

```
float[] xOriginal = {100, 140, 40, 160, 60};
float[] yOriginal = {40, 160, 80, 80, 160};
```

Note that these two arrays are the same size. At bin `0`, we have the `x,y` for point `0`, at bin `1` we have the `x,y` for point `1`, and so on.

Then, let's setup our `draw` loop to draw them. Note that we need five lines, and use modulo to wrap around the last point to 0.

```
for (int i = 0; i < xOriginal.length; i++)
{
   int next = (i+1)%xOriginal.length;
   line(xOriginal[i], yOriginal[i],
        xOriginal[next], yOriginal[next]);
}
```

Add code to make each point move by a random amount up to some maximum. I have my max set to 3 in a variable called WANDER. Make sure to do this inside the for loop, for each point. You could use a separate for loop for this, but we'll combine for now to save work:

```
for (int i = 0; i < xOriginal.length; i++)
  {
    int next = (i+1)%xOriginal.length;
    line(xOriginal[i], yOriginal[i],
         xOriginal[next], yOriginal[next]);
    xOriginal[i] += random(2*WANDER)-WANDER;
    yOriginal[i] += random(2*WANDER)-WANDER;
  }
```

If you make sure to setup properly, clear the background, and set the stroke color, you should see a 5-pointed star that wanders around.

However, we need to add the functionality of setting the points back to their original. In our current version, we modify the original points so they are lost. What we need to do, instead, is to first make a copy of the points, and work on those. That way, we still have the original ones for use later.

Let's add the two new arrays and instantiate them with the proper size, at the top of the program:

```
float[] xPoints = new float[xOriginal.length];
float[] yPoints = new float[yOriginal.length];
```

This will serve as our copies.

First, create a void function that copies those original points, verbatim, into the new arrays. This will make the new arrays have the exact same points as the originals. Since we already instantiated the copy arrays, all we need to do is to copy all the data from the bins straight over, like we did in the previous section.

```
void copyOriginal()
{
  for (int i = 0; i < xPoints.length; i++)
  {
    xPoints[i] = xOriginal[i];
    yPoints[i] = yOriginal[i];
  }
}
```

We can now use this to copy the original points into our new arrays. Let's first do this on program startup, and also when the mouse is pressed.

The next piece of the puzzle is that the `draw` block needs to be updated so that, instead of using the original arrays, it uses the copies. This way, we are drawing and moving the copied points, while not touching the originals. When the mouse is pressed, we re-copy the originals into the copies so that it starts over.

Here is the whole program.

```
final float WANDER = 3;

float[] xOriginal = {100, 140, 40, 160, 60};
float[] yOriginal = {40, 160, 80, 80, 160};
float[] xPoints = new float[xOriginal.length];
float[] yPoints = new float[yOriginal.length];

void setup()
{
  size(500, 500);
  copyOriginal();
}

void copyOriginal()
{
  for (int i = 0; i < xPoints.length; i++)
  {
    xPoints[i] = xOriginal[i];
    yPoints[i] = yOriginal[i];
  }
}

void draw()
{
  background(0);
  stroke(255);
  if (mousePressed)
    copyOriginal();

  for (int i = 0; i < xPoints.length; i++)
  {
    int next = (i+1)%xPoints.length;
    line(xPoints[i], yPoints[i],
         xPoints[next], yPoints[next]);

    xPoints[i] += random(2*WANDER)-WANDER;
    yPoints[i] += random(2*WANDER)-WANDER;
  }
}
```

Check your Understanding

17.7 Check Your Understanding: Exercises

Exercise 1. Does the following code return `true` or `false`?

```
float[] a = {100, 140, 40, 160, 60};
float[] b = {100, 140, 40, 160, 60};
println(a==b);
```

Exercise 2. Using the `==` operator on arrays and strings does not compare the array or string, but it is still correct code. When would you want to use this? Why is it even allowed?

Exercise 3. What is wrong with the following code?

```
float[] a = {100, 140, 40, 160, 60};
a = null;
println(a[0]);
```

Using your knowledge of arrays and memory, explain what happens.

Exercise 4. Given some array of integers `a`, make a new array `b` that is double the size of `a`, and has the data from `a` in the first half of its bins (all of `a`).

Exercise 5. Implement a series of special array comparison techniques that compares two arrays, `a` and `b`.

a. Given two integer arrays, they are considered equal if a) they have the same length, and b) the data in them sum to the same total.
b. Given one integer array and a floating point array, they are considered equal

if a) they have the same length, and b) the floating point values cast to integers equal the values in the integer array.

c. Given one integer array and one boolean array, they are considered equal if a) they have the same length, and b) an even number corresponds to true, and an odd number corresponds to false.

Exercise 6. Make a program that generates 10 random points on startup, and stores it in two arrays (one for `x` and one for `y`). In the `draw` block, put ellipses at each point in random colors.

a. If the mouse is pressed, double the number of points, adding new random ones. As the array size changes, it should not lose its data, and, the rest of the code should work properly.
b. If the keyboard is pressed, halve the number of points, removing every second point.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)

UNIT 18. ARRAYS AND FUNCTIONS

Summary

You will learn about how arrays can be used with functions. Specifically

- Basic syntax for using arrays with functions
- How arrays are stored in memory impacts how they are used with functions
- How arrays can be modified inside functions

Learning Objectives

After finishing this unit, you will be able to …

- Use arrays as parameters to functions
- Use arrays as return types from functions
- Create a new array inside a function
- Modify the data in an array in a function

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

18.1 **Introduction**

On the surface, arrays and functions are straightforward and nothing new. Array types can be used like any other type, so we can use it as a return type or parameter:

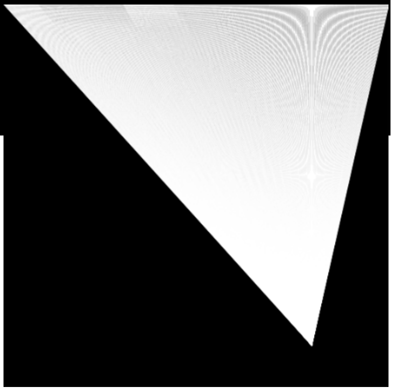
```
type[] someFunction(type[] someVariable) {…
```

For example, let's make a function to generate an array of size `n`, with the numbers `1..n` in the bins. The header looks like

```
int[] makeArray(int n)
```

Once inside the function, we make a new array, use a for loop to fill it with the numbers `1..n`, and return it. Watch out for an off-by-one error, i.e., bin `0` has `1`, bin `1` as `2`, …, bin `i` has `i+1`, … bin `n-1` has `n`.

```
int[] makeArray(int n)
{
  int arr = new int[n];
  for (int i = 0; i < arr.length; i++)
  {
    arr[i] = i+1;
  }
  return arr;
}
```



We can test this in our program, e.g., we can generate the array once, in our setup, and then draw from those points on the top border (with the array values being the `x` coordinate) to the mouse.

First the setup just calls `makeArray` and stores the result in a global array:

```
void setup()
{
  size(500, 500);
  points = makeArray(width);
}
```

Then, the draw uses a `for` loop to go through each bin, and draw a line from `(points[i], 0)` to the mouse:

```
for (int i = 0; i < points.length; i++)
{
  line(points[i], 0, mouseX, mouseY);
}
```

Similarly, we can send arrays into functions as parameters. Let's modify the above code by making a function that takes an array of `x` values, and draws points from those `x` values, at the `y` origin, to the mouse. Move the code out of the `draw` block. The function header is as normal – just make the variable to be an array type:

```
void drawLines(int[] xValues)
{
  for (int i = 0; i < xValues.length; i++)
  {
    line(xValues[i], 0, mouseX, mouseY);
  }
}
```

Now like any function, this is a reusable piece of code. You can toss in any array variable, and it will use that data to draw the lines, since the code only works on the parameter local variable, and not the global.

18.2 **Arrays, Functions, and Memory**

Although the above example was straight forward, things get tricky very quickly because array variables only store the memory address, not the whole array. Consider this example:

```
int[] numbers = new int[1];
void setup()
{
    numbers = makeArr(10);
    numbers[9]++;
}
int[] makeArr(int n) {
    int [] data = new int[n];
    for (int i = 0; i < data.length; i++) {
      data[i] = i+1;
    }
    return data;
}
```

What happens? Is this okay? We start by making the array with only one bin in it. Yet, later, we are accessing the 10th bin (bin 9).

Actually, this works. What is going on?

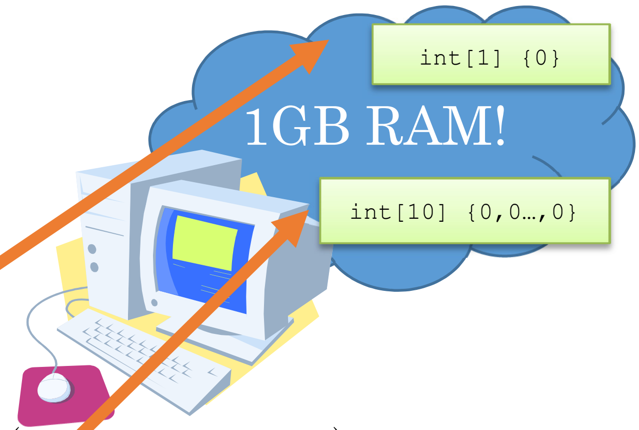
The trick is to realize that the array variable only holds the memory address, and, this can change. We can peek at this (just for learning purposes) by asking Processing to toss out the memory addresses.

Modify your setup block as follows:

```
void setup()
{
    println(""+numbers);
    numbers = makeArr(10);
    println(""+numbers);
    numbers[9]++;
}
```

If you run it now, even though the memory addresses themselves don't make any sense, you can see that they are different. The address changed after the call to the `makeArr` function.

Let's think about that visually.



(modern looking computer)

First, when the program starts, we make a new array of size 1, and it gets filled with the default values. The variable `numbers` only holds this address.

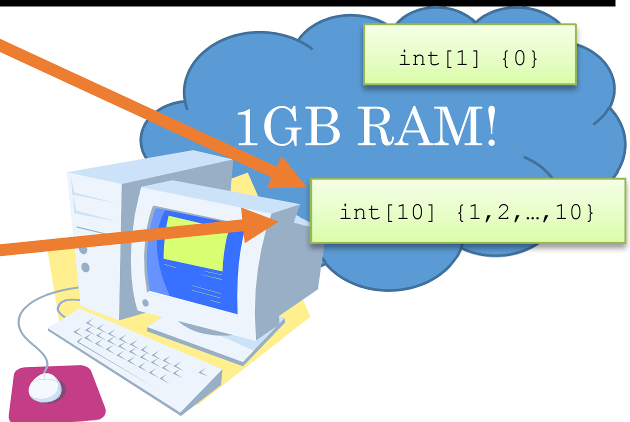
```
int[] numbers = new int[1];
```

Later, we call the `makeArr` function with the number 10 as a parameter. First, it makes a new array in memory of size 10 and fills it with 0s.

```
int [] data = new int[n];
```

Then it fills this array with data, from 1 to 10:

```
for (int i = 0; i < data.length; i++)
{
  data[i] = i+1;
}
```



(modern looking computer)

Finally, the function returns the data array. However, in this case, it actually only returns the memory address of the array, since that is what is stored in the variable.

```
numbers = makeArr(10);
```

And then it stores that address in our `numbers` variable. Previously, `numbers` pointed at the `int[1]` as in the diagram on the previous page. However, now `numbers` points at the new array. Unfortunately, no one is pointing at the old array and it is lost forever!

Now, since `numbers` points to an array of size 10, the code for accessing bin 9 works.

Remember: Array variables only store the memory addresses!

This quirk of arrays makes things quite complicated when dealing with functions.

18.3 **Example: generating Fibonacci numbers**

Let's make a function that takes an integer, `n`, and returns an array. It does the following:

- Makes a new array of size `n`
- Fills the array with the first `n` Fibonacci numbers
- Returns the new array

As a reminder, the Fibonacci number sequence is defined as:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

For example, the first 6 numbers are 0, 1, 1, 2, 3, 5 … First, we setup the function header. It simply takes an integer, `n`, and returns the new array:

```
int[] fibonacci(int n)
```

Then, we need to use a for loop to calculate the numbers. First, however, we need to setup the initial conditions, the F_0 and F_1.

```
f[0] = 0;
f[1] = 1;
```

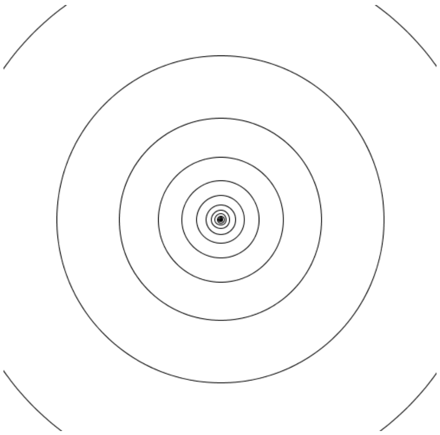
Warning: there is a danger here. What if we only want the first number only? If `n` is 1? Then the second line will crash. We can use `if` statements to make this safe:

```
if (n>=1)
   f[0] = 0;
if (n>=2)
   f[1] = 1;
```

Now that we setup the first 2, we can generate from the third one on using a `for` loop. At each bin, simply look at earlier bins to calculate the new number, as in the formula on the prior page:

```
for (int i = 2; i < n; i++)
{
  f[i] = f[i-1] + f[i-2];
}
```

And we're done! Return the array. Remember that we are supposed to make a new array here, so this instantiates the array. Wherever you use this, you just need to declare the array variable, and do not need to instantiate it elsewhere.



Let's use this function to visualize what this sequence looks like: draw circles with radius set to the Fibonacci numbers. Since this is not interactive, let's do everything in the `setup` and forget the `draw`. After setting the screen size and clearing, we first create the array into a local variable (I used a global to set the number of Fibonacci numbers):

```
int[] fib = fibonacci(COUNT);
```

NOTE: Wait a second – why am I just creating the array variable, `fib`, but not instantiating the array? We always need to instantiate the array! The answer is that the array is instantiated inside the fibonacci function. It calls new to create the array, and returns the address to store in the fib variable.

Then, we just run a for loop over the numbers, and draw circles at the center of the screen with the radius equal to the Fibonacci number:

```
for (int i = fib.length-1; i >=0; i--)
{
  fill(255);//random(255));
  ellipse(width/2, height/2, fib[i]*2, fib[i]*2);
}
```

18.4 **Modifying an Array in a Function**

First, consider this example without arrays, that changes a regular variable value in a function

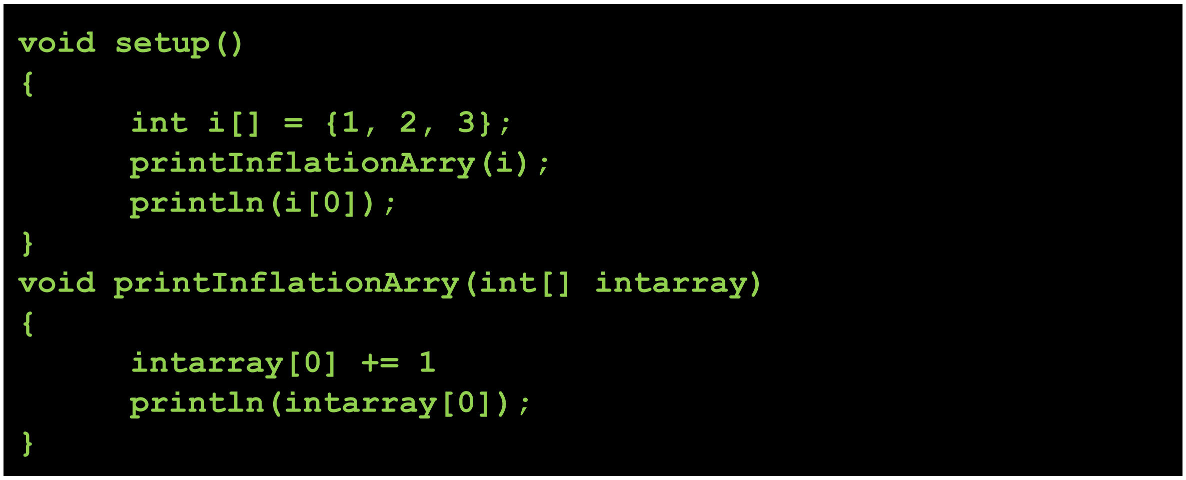
```
void setup()
{
      int i = 15;
      printInflation(i);
      println(i);
}
void printInflation(int number)
{
      number += 1;
      println(number);
}
```

What is the output here? The output is

```
16
15
```

In setup, i is set to 15. printInflation is called, and 15 is copied in and stored inside the local variable number. Inside printInflation, 1 is added to number, which becomes 16, so 16 is printed out. Once the function returns to the setup block, no data was returned. Since only a copy of i was passed into the function, the original i stays unchanged at 15. So, 15 is printed out second.

By now this should be clear to you. However, let's do something similar with arrays:



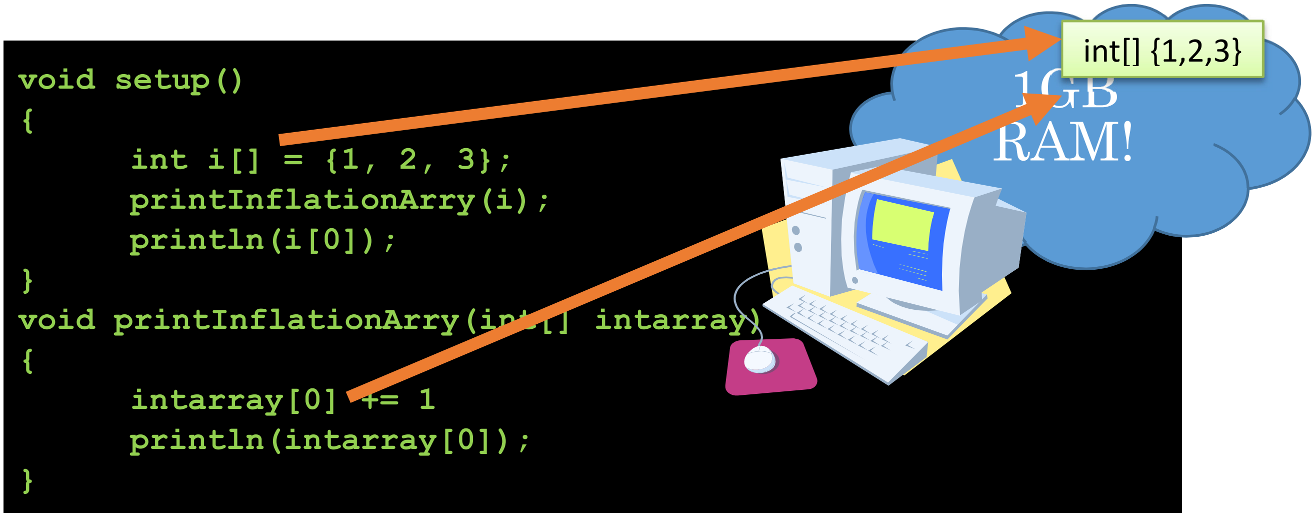
```
void setup()
{
      int i[] = {1, 2, 3};
      printInflationArry(i);
      println(i[0]);
}
void printInflationArry(int[] intarray)
{
      intarray[0] += 1
      println(intarray[0]);
}
```

What is the output? If you run the above code, you get 2 and 2.

Wait a second – why is this? This example looks just like the previous one – we change something inside a function, but, it gets changed back in the original, too! What happened?

This comes back to the fact that array variables only store the address of the array and not the whole array. When we call `printInflationArry`, we pass the address along to the original array. ***The address gets copied, the array does not get copied.*** There is only one array.

So when `printInflationArry` gets a copy of the address, and it modifies the array, it modifies the original. It prints out the modified bin, and after it returns, the `setup` block also prints out the same bin, since they are working on the same array.



```
void setup()
{
      int i[] = {1, 2, 3};
      printInflationArry(i);
      println(i[0]);
}
void printInflationArry(int[] intarray)
{
      intarray[0] += 1
      println(intarray[0]);
}
```

As you can see, there is only ever one array created. Initially, `i` gets set to the address of this new array. When `printInflationArry` is called with `i`, this address gets copied in. Inside the function, the `intarray` variable points to the same array as we used in the `setup` block. When the code modifies the array pointed to by that address, the single array changes. The array pointed to by `i` back in setup changes.

This actually works exactly the same as before. We copy the memory address in. The function cannot change what is in the variable `i`. However, since it has the address to the array, it can change that.

Here is a similar example. What is the output from the code below?

```
void setup() {
      int i[] = {1, 2, 3};
      makeNewArray(i);
      println(i[0]);
}
void makeNewArray(int[] intarray) {
      intarray = new int[3];
      intarray[0] = 5;
      println(intarray[0]);
}
```

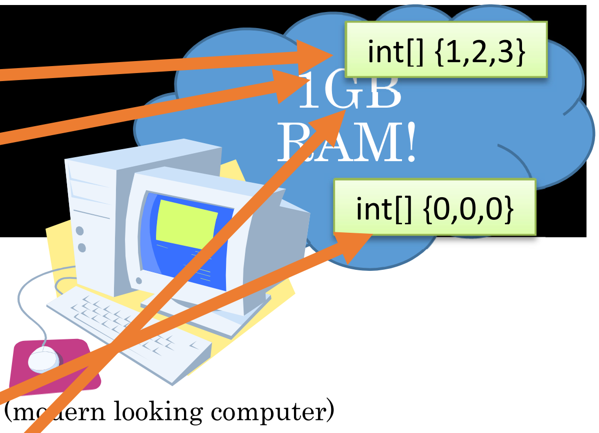
The output is actually

```
5
1
```

Why does this happen? This seems to contradict the previous example, where changes made to an array in a function, reflect back in the calling function. Again, remembering that array variables only store the memory address helps solve this mystery.

What is happening is that, even though we pass the address of `int[] i` from `setup` into `makeNewArray` (which is then copied into `intarray`), the `makeNewArray` function immediately throws away that memory address, and gets a new one: it calls `new int[3]`, which makes a new array, and stores that new memory address in `intarray`. We modify that new array. Back in setup, the `i` variable still points to the old array. The new memory address we got in `makeNewArray` stayed there, and it had no way to get back. Let's look at this visually:

```
void setup() {
      int i[] = {1, 2, 3};
      makeNewArray(i);
      println(i[0]);
}
```



In setup the new array is made in memory as usual, and the address is stored in `i`. When `makeNewArray` is called, this address is copied in.

```
void makeNewArray(int[] intarray) {
      intarray = new int[3];
      intarray[0] = 5;
      println(intarray[0]);
}
```

Inside `makeNewArray`, initially `intarray` points to our single array in memory. Immediately following, we call `new` to create a new array in memory, so now `intarray` points to the new array. We modify the new array in place (set the first bin to 5), and print out that bin in the new array. Notice how `i` in `setup` still points to the original, unchanged array, so when we return to setup, we print the bin from the original array. The address from the new array is lost, and we can no longer access it once the function is over.

18.5 **Example: moving a point field around**

Let's do an example which highlights many of the points of tossing arrays around with functions. We will create an array in a function, modify an array in a function, and send multiple arrays to a function.

This program will draw a collection of random points around the screen. Using the keyboard you can move the whole lot around (like moving a piece of paper). Clicking the mouse gives a new set of random points.

First, create globals to store your x and y points. For now, let's just create the array variables, we will instantiate them later:

```
int[] pointsX;
int[] pointsY;
```

To create these arrays and populate them with data, we need a function that will create an array and fill it with random values within a specific range. In this case, let’s make a function that takes the array size, and the maximum value, and returns such an array. Then, we can use it to generate both the x and the y values.

```
int[] newRandomArray(int n, int max)
{
  int[] a = new int[n];
  for (int i = 0; i < a.length; i++)
  {
    a[i] = (int)(random(max+1));
  }
  return a;
}
```

Keep in mind that this function does not modify any global variables. It creates a new array, populates it with values, and returns the address to this new array. Hmm... Why does it add +1 to the max value? Now, in our setup block, we can call this function to generate our random x and y points:

```
void setup()
{
  size(500, 500);
  pointsX = newRandomArray(COUNT, width);
  pointsY = newRandomArray(COUNT, height);
}
```

This is important: here, we only create the variable at the beginning of the program. The arrays are created (instantiated) inside the newRandomArray function. It returns a memory address pointing to a new, random array, that we store in our global variables. From that point forward, those globals point to the random arrays.

Next, make a function that takes two integer arrays (one for x, one for y), and draws the points on the screen. This should be straight forward, but keep in mind that the memory address is copied in. No matter where the arrays were created, as long as this function receives memory addresses to two arrays, it is happy.

```
void drawPoints(int[] x, int[] y)
{
  stroke(255);
  for (int i = 0; i<x.length; i++)
    point(x[i], y[i]);
}
```

Next, let's make a function that modifies an array. It takes an array, and an integer, and adds the number to each bin. Since only a memory address comes in, whatever changes are applied to the array inside this function, are permanent, and will be reflected back in the array that was used to call the function. ***This is the case even though this function returns void. The function can modify the array since it knows the address to it***.

```
void addToArray(int[] data, int value)
{
  for (int i = 0; i < data.length; i++)
    data[i] += value;
}
```

Finally, let's work up the `draw` block. The basics are easy: clear the background, and draw the points using `drawPoints`. We need some additional (slightly tedious) logic to check which key or mouse is pressed. Depending on the key, we either add `1` or `-1` to `pointsX` (to move right or left), or similarly `1` or `-1` to `pointsY` (to move down or up). The thing to remember here is that, although those `addToArray` calls do not have a return value, since we are passing the array address, we can expect that the array is modified inside the function. This contradicts what is possible with regular variables.

To make this happen, we will introduce a new technique: how to tell which key was pressed. You may have encountered this already, for example, in an assignment or reading on your own.

While we know that we can use `keyPressed` to tell if a key is currently down on the keyboard, how do we know which key was pressed? Luckily, Processing provides global variables that contain the last keys pressed. You can read more about this on the Processing website, but some keys are coded and do not provide characters, such as the arrow keys, the enter key, etc. We can check the code of the last key pressed using the `keyCode` global. Further, Processing provides some constants that know the codes of typical keys. With this in mind, the following code should make sense to you:

```
void draw()
{
  background(0);
  if (keyPressed)
  {
```

```
  if (keyCode == UP)
   addToArray(pointsY, -1);
  else if (keyCode == DOWN)
   addToArray(pointsY, 1);
  else if (keyCode == RIGHT)
   addToArray(pointsX, 1);
  else if (keyCode == LEFT)
   addToArray(pointsX, -1);
 }
 if (mousePressed)
 {
  pointsX = newRandomArray(COUNT, width);
  pointsY = newRandomArray(COUNT, height);
 }
 drawPoints(pointsX, pointsY);
}
```

Finally, it is worth discussing what happens with the mouse is pressed. When the `newRandomArray` function is called twice, it generates brand new arrays with new memory addresses. Those addresses are returned, and stored in `pointsX` and `pointsY`. The old addresses are lost forever (and the old arrays abandoned). Anywhere in the program that uses `pointsX` and `pointsY` from this point will be using the new arrays.

Check your Understanding

18.6 **Check Your Understanding: Exercises**

Exercise 1. Make a function that takes an integer array, and returns a single integer. The function will add up all the values in the array, and return the total.

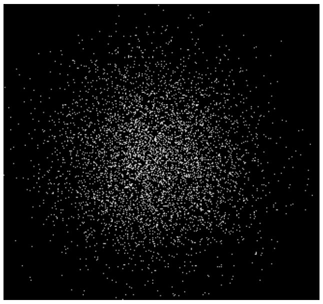
Exercise 2. Make a function that takes two integer arrays in as parameters and returns void. If the arrays are the same size, then swap every second element between the arrays. For example, given {1, 2, 3, 4} and {10, 11, 12, 13}, you should end up with {1, 11, 3, 14} and {10, 2, 12, 4}

Exercise 3. Make a function that takes in an array of floating point numbers as a parameter. It creates a new array of the same size, and copies the data from the input array into this new array, except in reverse order. The new array is returned.

a. Make a new function that takes two floating point arrays as parameters and returns true if they hold the same data (in the same order), and false otherwise.
b. Use your two functions to determine if a set of ordered numbers is a numerical palindrome, that is, the same forwards and backwards.

Exercise 4. Make a function called `subset` that takes two arrays of integers, `a` and `b`. The function returns true if, for every item in `a`, it is found in `b`. Conversely, if there exists an element in `a` that does not exist in `b`, return false.

Exercise 5. A big swarm of flies (did someone leave some banana peels in the garbage over the weekend??) You will make a bunch of points start at the screen center, and then move randomly. It looks like a (gross) swarm of flies.



The point of this lab is to use arrays with functions. As such, **do not use any globals – finals or otherwise – in your three functions you will make.** All data will be either passed into or out from the functions.

Make the following globals. `final int FLIES` determines how many flies are in the program. Try 500 to start. `final int MAX_MOVE` determines how much the flies move. I use 5. Make two integer array variables, `pointsX`, `pointsY`; just make the variables, do not instantiate or create the arrays. Make the following three functions:

`int[] newArray(int size, int value)`. Takes a size, makes a new array of that size, and sets every bin to value (use a `for` loop). Return the new array.

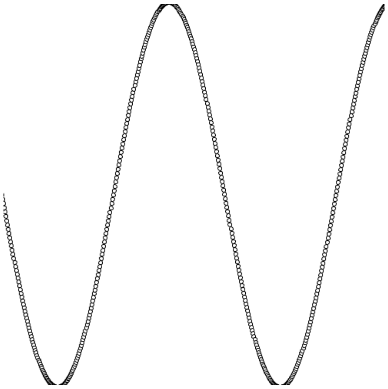
`void drawPoints(int[] x, int[] y, int clr)`. Assuming `x` and `y` are the same length, sets the draw color to `clr`, and draws all the points `x[i], y[i]`.

`void changeArray(int[] a, int maxChange)`. Goes through each bin in `a`, calculates a random change from `-maxChange` to `maxChange`, and applies it to that bin. Each bin should get a new random number applied.

In `setup`, set the canvas size, and call `newArray` twice to set your `pointsX` and `pointsY` to the screen center. Save in your global arrays.

In `draw`, clear the background, call `changeArray` twice (once on `pointsX` and once on `pointsY`), and call `drawPoints` once. Should work! Grab some chopsticks and catch those flies!

Exercise 6. You will plot the sine function, after tossing it into an array. Sometimes, to save memory or time, we use coarser approximations. In this lab, you will also learn how to approximate the function with a half or quarter the number of points.



You will only need globals for the number of points (set to 200), the ellipse size (I used 5), and the colors. You will need the following four functions.

`float[] sinArray(float min, float max, int steps)` – Creates a new array with `steps` bins, and fills it with the sine function from `min` to `max`. After making the new array, use a `for` loop to go through each bin. At each bin, calculate the angle (to use in `sin`) as `bin/(bincount -1)*(max-min) + min`. (make percentage from 0..1, multiply by the range, and add the minimum). Then use this angle in the `sin` function, and store the result in your array at the appropriate bin.

`float[] halfArray(float[] array)` – Creates a new array that is half the size of the one provided, with the data approximated. It samples the original array into the new one. Since the new array has half the bins, we take the pairs of bins in the old array and average them for our value. That is, for `0<=i<newarray.length` we set `newArray[i]` to be the average of `oldArray[i*2]` and `oldArray[i*2+1]`. Make sure to return the new array!

`float[] quarterArray(float[] array)` – Creates a new array that is a quarter of the one provided. We can do this simply by calling `halfArray` twice!

`void plotSine(float[] data, int clr)` – Plots the data on the screen. This one is annoying, but it is ok once you figure it out. First, make a `for` loop to go through each bin in `data`. You figure out `x` by taking the percent along the array `(i/(length-1))`, and multiplying it by the screen width. For `y`, since we know that sine goes from `-1..1`, and we want it centered, we take the `data[i] *height/2`, and add it to the center of the y axis. Draw the `ellipse`.

`setup` just sets the canvas size. `draw` loop, each time, should have the `min` at 0, set the max to be `mouseX/width*8PI`, create a new `sinArray` into a local variable, if the mouse is pressed half that array using `halfArray`, if a key is pressed quarter that array. Plot it regardless. The data is not saved (so lost) at the end of the `draw` function.

Exercise 7. Images! You will load two images – a full color one, and a corresponding black and white one. If the user pressed the mouse button, it will `draw` color on the black and white canvas.

There is a bunch of new stuff in this one. We will be using a new Object type called `PImage` (short for Processing Image). It can load images that are a part of your project. You can use your own images, but I provided two for you that I know work, so maybe start there. Check them on the website under Unit 18. You can add the images to your sketch by using Processing. Click `“Sketch”` menu, then `“add file”`, and add the images, one by one, to your project. To see what files are already added, click `“Show Sketch Folder”` to see.

Once you create a `PImage` variable,

```
PImage img;
```

Then you can use the following functions and methods that are built in:

`PImage loadImage(String name);` This loads the given file and returns a `PImage`, e.g., `img = loadImage(“photo.jpg”);`.

`void img.loadPixels();` This converts the internal image format to an array that you can access

`void img.updatePixels();` This updates the internal image based on the image array.

You also have the following variables accessible through the image object:

`int[] img.pixels;` This is an integer array representing the pixels. Useless until you call `loadPixels`. If you change this array, call `updatePixels` to make the image updated.

```
int img.width;
int img.height;
```

At the beginning of your program, create two `PImage` variables (`PImage img`, and `PImage bwImage;` for the color and black and white ones). Also, create two strings for the file names. I have `BW_IMG_NAME = "photobw.jpg"` and `IMG_NAME = "photo.jpg"` (the ones provided for download). These must refer to an image in your project.

So now we can load images, and, get integer arrays of their pixels. There is a gotcha – images are two-dimensional, yet we only get a one dimensional array. This is standard for images, and the pixels are ordered in a fixed order: the top left corner is index 0 of the array, and they increase as you move right. So, the first row is index 0..width-1. They just continue on the next line, from width..2*width-1. So, given an x and a y, the index into the array is x + y*width (see inset).

In your setup, call loadImage twice to get the two images and store the result in your global variables. Also, call loadPixels on both since we will be working with the pixel array.

y \ **X**	**0**	**1**	**2**	**…**	**w-1**
0	0	1	2	…	w-1
1	w	w+1	w+2	…	2w-1
2	2w	2w+1	2w+2	…	3w-1
…	…	…	…	…	…
…	yw	…	…	yw+x	…
h-1					wh-1

How pixels are numbered in an image, where **w** is the width and **h** is the height: numbering starts at the top left of the image and goes right, and continues on the next line. The pixel number can be calculated by yw+x, and given a pixel number n, x=n%w and y=n/w

Set the canvasSize to (img.width, img.height). It doesn't matter which of the two images you use since they should be the same size.

In this example, we have two images, and, we will copy parts from one image to another (from the color one to the b&w one). To do this, all we need to do is to copy array data from one array to the other.

Make the following functions:

int pixelIndex(int x, int y, int w). Converts x,y for width to a one dimensional array index.

float dist(int x, int y, int x2, int y2). Returns Euclidean distance between two points

void drawAtMouse(int[] toImg, int[] fromImg). This function copies all pixels from fromImg to toImg that are within a specified distance to the mouse. I used BRUSH_SIZE = 10. To do this, you setup a nested for loop to go through all x,y of the fromImage. (both images should be the same size). You calculate the distance between each x,y and the mouseX, mouseY. If it is less than your brush size, you call pixelIndex to get the index of this x,y, and copy this bin from fromImg to toImg. That's it! Make sure to use a nested for loop over the screen dimensions, as it makes it easier.

In draw, if the mouse left button is pressed (make sure to check if a button is pressed AND it's the left one), then call drawAtMouse with the two images. If the right mouse button is pressed, re-load the b&w image and make sure to call `.loadPixels` again (lets you start over). At the end of the draw block, call `.updatePixels()` on your black and white image to reflect any changes you may have made. Finally, you can draw the image with the background command – yep! You can do `background(bwImage);` and it draws the image, as long as the image is the same size as the canvas.

Exercise 8. You will implement a simplified base for the dice game Yahtzee™. For those familiar with the game:

- Only the top half of the score sheet will be used.
- It is for only one player.
- Only a simplified strategy will be allowed: The player must choose to keep all dice of one particular value, and re-roll the rest.

This will be implemented in steps.

- Step a) will only allow 5 dice to be rolled.
- Step b) will allow re-rolling of some of the dice.
- Step c) will keep score, and complete a simplified game.

The game is played by rolling 5 dice. The dice roll will be stored in an integer array containing 5 integer values from 1 to 6. We will leave most of the drawing the dice to screen and completing the user input to your own time. Here, we focus on the logic and array problems to be solved. Make sure to write good code in the `draw` block to test your functions.

a. Create the following functions:

   `int[] rollDice(int numDice)` create and return a reference to a dice roll – an array containing `numDice` integers from `1` to `NUM_SIDES`.

   `void showDiceRoll(int[])` draw the dice. For now, just draw to console, you can add graphics later.

b. The player is now allowed to make two additional rolls. The objective is to get as many as possible that are showing the same number. In the real game, the player can choose to keep any subset of the dice. In this version, the player can only choose one "goal" number. All dice showing that number will be left alone, and the rest will be re-rolled.

   Once this has been done twice, the player must add up their score, the sum of those dice showing the particular number from 1 to 6. For example, if the dice are showing 5-3-2-3-3 and the player chooses 3, the score will be 9 (add up the 3's only – the 5 and 2 are ignored). If the player chooses 2, the score will be 2. If the player chooses 1, the score will be 0.

   Create the following two small functions:

`void tryFor(int target, int[] dice)` accept a dice roll and a target number. Dice that are equal to the target should be left unchanged. All the rest should be re-rolled.

`int scoreAs(int number, int[] dice)` calculate the score for the given roll, if the player chooses the given number. All dice showing that number are added up, and all others are ignored.

c. To play the complete (still simplified) game, the player is given 6 turns. When a player chooses to score a particular number, for example 3's, the score will be entered beside "3's" on the scoresheet. Once a number is chosen, it cannot be chosen again. Each number must be used exactly once. A total score is calculated. If that total score is 63 or more, a bonus of 35 points is added to the score (once only).

A score sheet will be represented by an `int[]` array containing 8 values. The first six will be the scores for 1's to 6's. A value of `EMPTY` (predefined as -1) will indicate that the row is empty (has not yet been chosen). The last two rows will be for the Bonus score and the Total score. Use named constants `SCORE_SHEET_SIZE`, `BONUS_ROW`, and `TOTAL_ROW` to represent the number of rows in the score sheet (8), the index of the Bonus entry (6), and the index of the Total entry (7).

Create the two following small functions:

`void int[] newScoreSheet()` create a new blank score sheet. The entries for 1's-6's should be `EMPTY` (-1), and the entries for the bonus and total should be 0.

`boolean enterScore(int[] scoreSheet, int row, int score)` add another turn to the score sheet. `row` must be 0 to 5, `score` is the score for the turn. If `row` is currently empty, then return `true`, and update the score sheet properly (including adding the bonus if earned), otherwise return `false` and do nothing else.

d. Drawing graphics is quite time consuming and can be tedious. Also, we did not really cover advanced user input in this course. As such, moving from the above functions into an actual working game is not trivial. Here we have provided some code that gives these features, and interfaces with your existing code that you wrote on this question. Try to get it working, and, ask your instructor about any of the advanced techniques used. The code can be found on the course website, called `DiceGame.pde`.

e. For most of the scoring in the full game, it is necessary to know the number of times that each number (1 to 6) appears on the dice, and what the maximum number of times is (“3 of a kind”, “4 of a kind”, etc.).

The game also needs to know how long a “straight” you have. A straight is a series of consecutive numbers such as 1-2-3-4 or 2-3-4-5-6, in any order (4-5-3-6-2 is a straight, too). It also needs to know if you have a “full house” which is exactly 3 of one number, and 2 of another, such as 2-2-3-3-3 or 4-1-4-1-4.

Start with the file on the course website, `DiceGame2.pde`, which contains updated logic to use the following functions:

`int[] freqCount(int[] dice)` accept an array representing the numbers on the dice (an array of `NUM_DICE` values, each from `1` to `NUM_SIDES`). Create and return an array of `NUM_SIDES` elements indicating how many times each number from `1` to `NUM_SIDES` appeared. That is, count the frequencies of the dice sides. Note that since the indices of an array always start at 0, but dice values start at 1, the number of 4’s would be at index [3].

`int maxOfAKind(int[] freqs)` accepts an array of integers, and returns the biggest one. In terms of the game, if you give it the frequency count result from `freqCount`, it will tell you how many “of a kind” you have.

`int maxInARow(int[] freqs)` accepts an array of integers, and return the maximum number of consecutive non-zero values that it could find in the array. In terms of the game, if you give it the frequency count from `freqCount`, it will tell you how long a “straight” you have.

`boolean hasFullHouse(int[] freqs)` accepts an array of integers and determine whether or not one of them is a 3, and another one is a 2. In terms of the game, if you give it the frequency count from `freqCount`, it will tell you whether or not you have a full house.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

UNIT 19. WORKING WITH ARRAYS: TECHNIQUES

Summary

You will learn about new ways to use arrays when solving problems. Specifically:

- You will see two techniques for having arrays that are partially filled
- You will learn about searching, and two techniques for searching arrays to see if data is contained within it

Learning Objectives

After finishing this unit, you will be able to …

- Implement a count-as-you-go partially-filled array
- Implement an impossible-value partially-filled array
- Implement a linear search on an array
- Implement a binary search on an array

How to Proceed

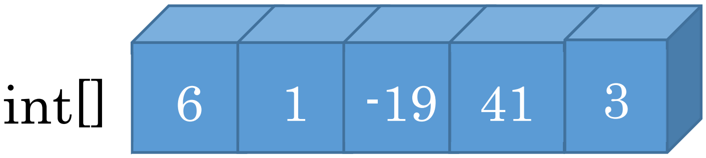
- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

19.1 **Introduction**

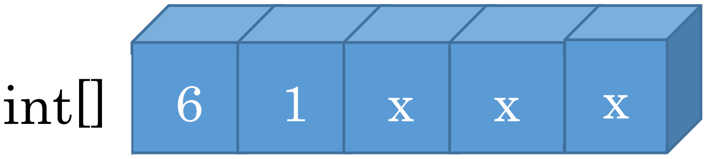
Until now in the course, each unit was specifically introducing a new programming structure, data type, technique, etc. However, you have already learned most of the basics of computer programming, and moving forward, more and more of your time will be spent on learning algorithms and data structures: ways to work with data using these tools. In this unit, we focus on learning some clever ways of working with data using arrays.

19.2 **Partially-Filled Arrays #1 – count as we go**

When we create an array, the default data is not usually that useful to us. Up until now, we have been fully populating our arrays with data before using them. That is, we set each bin to a desired value such as a position on the screen or a random value, and then start our program. For example, this is a fully populated, or fully-filled, array:

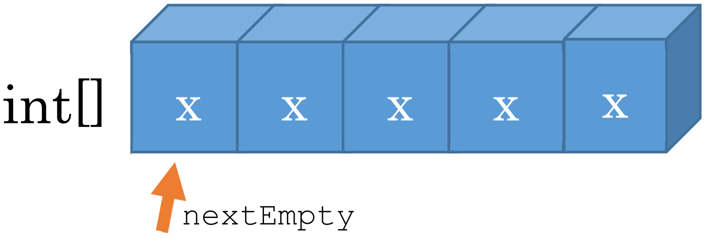


Sometimes, however, we want to have an array with data in some bins, but not in others. The most basic example of this is an array that fills up over time, but is not yet quite full:

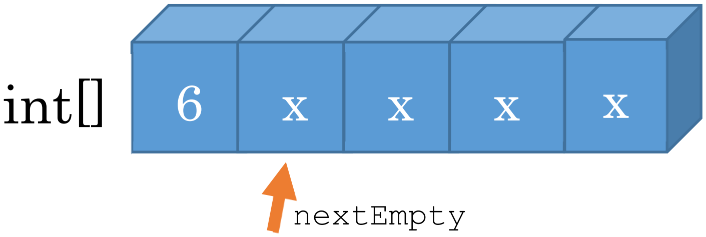


In this case, `x` marks an empty bin. The challenge in programming is to identify which bins have data, and which do not. Remember, with primitive data types there is no "no data" option (unlike `null` with objects and arrays). An integer must have a number, and with arrays, that number defaults to `0`.

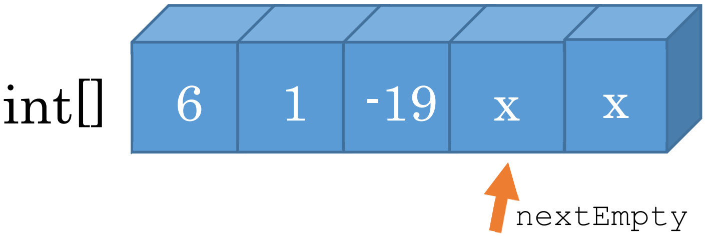
One technique is to always fill the array from the left, to the right, and use a separate integer variable to keep track of the next empty bin. At the beginning, the array is all empty, and our `nextEmpty` pointer is set to 0.



If we add a new value to the array, say a 6, then we put the data in the nextEmpty bin, and then move nextEmpty along:



And we keep going like this as we add data



We just need to be careful not to run off the edge of the array. When the array is full, nextEmpty will be equal to the array length. (it points to the first invalid bin number)

19.3 **Example: mouse path**

We will use an array to record a mouse movement as a series of points, and draw lines through the points. Unlike previous drawing examples, we store this in an array, which gives us power to modify or animate it. The array will start empty, and if we click, the current mouse positions get added to the array. It slowly fills up. If a keyboard key is pressed, we clear the path (empty the array).

We will need two partially-filled arrays – one for the x and one for the y coordinate. In my case, I will record up to 100 mouse positions.

```
final int HISTORY = 100;
int[] mousePathX = new int[HISTORY];
int[] mousePathY = new int[HISTORY];
```

In addition to the arrays to store the data, since they will be partially filled, we need a pointer to the next empty bin. At the beginning, the entire arrays are empty, so this points to bin 0:

```
int nextEmpty = 0;
```

Actually using these partially-filled arrays requires some changes. To add some data to them, we add the data at the next empty spot, and then move that along. In this case, if the mouse button is pressed, let's put the mouse X and Y into the next spot in the array, and move the pointer on to the next empty spot.

```
mousePathX[nextEmpty] = mouseX;
mousePathY[nextEmpty] = mouseY;
nextEmpty++;
```

In addition, before doing this, be sure to check that we are not running off the edge of the array. That is, only do this if `nextEmpty` is less than the array length. Otherwise, don't add anything.

Now, let's implement a function to draw. We have seen this before – it draws lines between the points. In this case, start from point 1, and draw a line back to the previous point, 0. Do this in a `for` loop, such that we are drawing from point `i` to `i-1`.

There is a catch here – how many lines do we draw? Previously, we just looked at the array size to setup our loop. However, here, we only want to include valid bins, so we need to know that number – we don't want to go over the entire array. In our case, the `nextEmpty` variable tells us this information. That is, if `nextEmpty` is 0, we have no data. If it's 5, we have 5 pieces of data (in bins 0...4). So, our draw function needs this information. Here, `clr` is a global variable that can change, initially set to white.

```
void drawPath(int[] x, int[] y, int bins)
{
  stroke(clr);
  for (int i = 1; i < bins; i++)
  {
    line(x[i-1], y[i-1], x[i], y[i]);
  }
}
```

When we call this in our draw block, we give `nextEmpty` as the number of bins.

Finally, we need a way to start over. Particularly given that our drawing stops if our arrays get full. How do you empty out these partially-filled arrays? All that needs to happen is for the `nextEmpty` pointer to be reset to 0, the beginning of the arrays. There is no need to empty the data out of the bins. When using the array, we use the pointer to determine how much is full. When putting data in the array, we use the pointer to determine where to put it. If a key is pressed, reset the array:

```
if (keyPressed)
{
  nextEmpty = 0;
}
```

That’s it! We used a partially filled array to hold a bit of mouse history. Ahead of time we don’t know how much we need to store so we make an array large enough, and only fill what we need. Here is my final code. I also added a function that the color used to draw changes if you press the keyboard, to illustrate that the entire path gets drawn every frame.

```
final int HISTORY = 100;
int[] mousePathX = new int[HISTORY];
int[] mousePathY = new int[HISTORY];
int nextEmpty = 0;
int clr = 255;

void setup()
{
  size(500, 500);
}

void drawPath(int[] x, int[]y, int bins)
{
  stroke(clr);
  for (int i = 1; i < bins; i++)
  {
    line(x[i-1], y[i-1], x[i], y[i]);
  }
}

void draw()
{
  background(0);
  if (mousePressed)
  {
    if (nextEmpty < mousePathX.length) {
      mousePathX[nextEmpty] = mouseX;
      mousePathY[nextEmpty] = mouseY;
      nextEmpty++;
    }
    clr = (int)random(128)+128;
  }
  if (keyPressed)
  {
    nextEmpty = 0;
  }
  drawPath(mousePathX, mousePathY, nextEmpty);
}
```

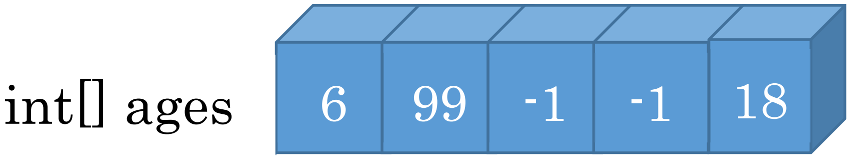
19.4 **Partially-Filled Arrays #2 – impossible values**

Sometimes, we cannot rely on an array filling up in a nice, neat order from 0 upward. What if we want to add data in the middle of the array? Or delete data in the array, leaving a hole with an empty bin? In this case, ***we can use a different technique: set the empty bins to an impossible value.***

What constitutes an impossible value? Well, it depends heavily on the context. If you are working with people's ages, `-1` is impossible. If you are working with car speeds (in km/h), maybe `1000000` is an impossible value. The general strategy is to name it as a final constant, and use that name throughout your program. In Processing, when dealing with basic canvas coordinates, `-1` is a nice impossible coordinate value, although this wouldn't work for the 3D examples we have seen.

Advanced: Another common strategy for impossible values is to use the largest or smallest possible value for your data type. When you have data that wants to use a very wide and unpredictable range, this is a good strategy as it leaves as much open room for data as possible. Most languages provide named constants for this to avoid errors. For example, in Java and Processing you have `Integer.MAX_VALUE;` and `Integer.MIN_VALUE;`. No matter what approach you use, always make sure to check for the impossible value, to avoid using it by accident resulting from a calculation. Another common approach is to use a *parallel* array – a second array, of Booleans, of exactly the same size. In this case, given data array `data` and boolean array `isEmpty`, then `data[i]` has data if `isEmpty[i]` is false.

This is how such a partially-filled array may look, e.g., for student ages:



Here, bins 2 and 3 are empty, with the other bins having data.

How would we ever end up in such a situation? That we have empty bins in the middle of an array? There are several ways. One is, what if we have an array of students, and one drops the course half way? Their bin becomes empty. What if each bin corresponds to a physical seat number in the room? Not all seats are filled, so some must be empty.

When working with these kinds of arrays, everything changes. When we want to add data to the array, we need to find an empty spot. If we want to print out the values, we have to skip over the empty bins. And so on. This takes more work, but is more flexible, as we can add or remove data at any point.

19.5 **Example: updated mouse path**

Let's change our previous mouse path to use our new partially-filled arrays technique. We will change from a path to drawing ellipses. Also, we will introduce the capability to delete points in the history – any ellipse under the mouse – to show how bins can be emptied and filled.

This will require a great deal of changes. Start by copy-pasting the previous example into processing. We will need to update everything to work with this new method. We no longer have a pointer to the next empty bin that we can rely on. After making a global for your `EMPTY` marker (set to `-1`, an impossible canvas coordinate), let's make a couple of helper functions. Let's make a function to clear an array – set all bins to empty – and do this at the beginning of the program to both the `x` and `y` arrays. (I renamed the `mousePath` to `mousePoints` for this example)

```
void setup()
{
  size(500, 500);
  clearArray(mousePointsX);
  clearArray(mousePointsY);
}

void clearArray(int[] a)
{
  for (int i = 0; i < a.length; i++)
    a[i] = EMPTY;
}
```

Now, we need a function that finds us the next empty bin so that we know where to add data. Previously we relied on a pointer that points to the end of the data in the array. Now, however, we need to go searching for one. Each time we want to store data in the array, we need to search out an empty bin, and use that one. It could be anywhere! We need a function to this. It takes an array, and tell us the index of the very first empty bin from left to right (from bin 0 up).

All that we have to do is to go through the array from the beginning until we find an empty bin. If none are empty, return -1 (an impossible bin value). I gave this a global name NOT_FOUND.

```
int nextEmpty(int[] a)
{
  int next = NOT_FOUND;
  for (int i = 0; i < a.length && next == -1; i++)
  {
    if (a[i] == EMPTY)
      next = i;
  }
  return next;
}
```

Notice how in this example I modified the for-loop conditional to quit early if we actually found an empty bin.

Next, let's update our drawPath to now be drawEllipses. Previously, we only went up until our nextEmpty counter. With our new technique, we actually need to go through the entire array. The difference is, we need to make sure that each bin is not empty before we use it, simply with an if statement.

```
void drawEllipses(int[] x, int[] y)
{
  stroke(255);
  for (int i = 0; i < x.length; i++)
  {
    if (x[i] != EMPTY)
      ellipse(x[i], y[i], 10, 10);
  }
}
```

Now let's update the draw. First, change the logic from using the old nextEmpty pointer, to searching for an empty spot and using it. You still have to check if the array is full! We know that it is full if there is no empty bin available. Also, notice how we only need to check one of the arrays, the x in this case: both the x and y arrays should be matched perfectly, so an empty bin in one matches an empty bin in the other.

```
if (mousePressed)
  {
   int empty = nextEmpty(mousePointsX);
   if (empty != NOT_FOUND)
   {
     mousePointsX[empty] = mouseX;
     mousePointsY[empty] = mouseY;
   }
   clr = (int)random(128)+128;
  }
```

Now, to erase the whole array, instead of resetting the `nextEmpty` pointer, we now actually need to go through and set each bin to empty. Luckily, we have functions to do this for us already:

```
if (keyPressed)
{
  clearArray(mousePointsX);
  clearArray(mousePointsY);
}
```

Next, let's start implementing the functionality to delete ellipses. When the key is pressed, instead of starting over, let's erase any ellipses under the mouse. This poses two problems: we need to find which ellipses to delete (under the mouse), and two, we need to actually delete them from the array.

To find the ellipse, we need to go through the arrays of points, calculate the distance between the mouse and that point, and if the distance is less than the radius of the circle, it's a hit.

First make a function to return the distance between two points (we have done this before). Them, make another function `erasePoints` that goes through the array, and erases any point that is less than the ellipse radius away (under the mouse). To delete the point, all we have to do is to set the bin values to `EMPTY`. As with the other functions, at each bin, don't forget to check if it's empty!! If it is empty already, don't do the distance calculation.

```
float dist(int x, int y, int x2, int y2)
{
  float diffX = x2-x;
  float diffY = y2-y;
  return sqrt(diffX*diffX+diffY*diffY);
}

void erasePoints(int[] x, int [] y, int underX, int underY)
{
  for (int i = 0; i < x.length; i++)
    if (x[i] != EMPTY)
      if (dist(x[i], y[i], underX, underY) <= ELLIPSE_SIZE/2)
      {
        x[i] = EMPTY;
        y[i] = EMPTY;
      }
}
```

Make sure to update your `draw` to use this new function. Now, you can draw points by clicking the mouse, and erase by using the keyboard. Here is the final example:

```
final int HISTORY = 100;
int[] mousePointsX = new int[HISTORY];
int[] mousePointsY = new int[HISTORY];
final int EMPTY = -1;
final int NOT_FOUND = -1;
final int ELLIPSE_SIZE = 10;
int clr = 255;

void setup()
{
  size(500, 500);
  clearArray(mousePointsX);
  clearArray(mousePointsY);
}

void clearArray(int[] a)
{
  for (int i = 0; i < a.length; i++)
    a[i] = EMPTY;
}

int nextEmpty(int[] a)
{
  int next = NOT_FOUND;
  for (int i = 0; i < a.length && next == -1; i++)
  {
    if (a[i] == EMPTY)
      next = i;
  }
  return next;
}

void drawEllipses(int[] x, int[] y)
{
  stroke(255);
  for (int i = 0; i < x.length; i++)
  {
    if (x[i] != EMPTY)
      ellipse(x[i], y[i], ELLIPSE_SIZE, ELLIPSE_SIZE);
  }
}

float dist(int x, int y, int x2, int y2)
{
  float diffX = x2-x;
  float diffY = y2-y;
  return sqrt(diffX*diffX+diffY*diffY);
}

void erasePoints(int[] x, int []y, int underX, int underY)
{
  for (int i = 0; i < x.length; i++)
    if (x[i] != EMPTY)
      if (dist(x[i], y[i], underX, underY) <= ELLIPSE_SIZE/2)
      {
        x[i] = EMPTY;
        y[i] = EMPTY;
      }
}

void draw()
{
  background(0);
  if (mousePressed)
  {
    int empty = nextEmpty(mousePointsX);
    if (empty != NOT_FOUND)
    {
      mousePointsX[empty] = mouseX;
      mousePointsY[empty] = mouseY;
    }
    clr = (int)random(128)+128;
  }
  if (keyPressed)
  {
    erasePoints(mousePointsX, mousePointsY, mouseX, mouseY);
  }
  drawEllipses(mousePointsX, mousePointsY);
}
```

We looked at two different methods for partially-filled arrays. ***The second method is much more flexible, we are able to easily delete things anywhere and insert them anywhere in the array. However, it came at a cost – we had to do a lot more work.*** Previously, to clear the array, we just set the `nextEmpty` pointer to 0 (1 operation). Now, we need to set EVERY bin to zero (`n` operations, one per element in the array). There is also a lot more work to put an item into the array, since we need to find a bin first. In a later course, you will study this kind of analysis, regarding the **complexity** of your algorithm. There is not always a right answer, it is usually a balance of trade-offs – sometimes you use more memory to save time, or do more work to save memory, etc. As a computer scientist, your job will know which tools and techniques to use in a given situation.

19.6 **Example: Automatically Grow Array**

Let's extend the above example. One limitation of our program is that we quickly run out of array memory. One solution would be to simply make the array very large at the start – but what if we don't need it in the end? This is a waste of memory. Instead, let's make the array grow when it fills up.

How can you resize an array? Short answer – you can't! The only way to do it is to make a new array with double the size of the old one, and copy the old data over. Let's make a function called `doubleArray` which:

- Takes an array as a parameter
- Makes a new array with double the number of bins

- Copies the data from the old array to the new array
- Sets the additional bins to empty
- Returns the new array

With this function, if we run out of memory we can simply just double the array again. Make sure to try and implement this yourself. It gets tricky as you have two array boundaries – the old one, and the new one. Here is my code:

```
int[] doubleArray(int[] oldA)
{
  int[] newA = new int[oldA.length*2];
  for (int i = 0; i < oldA.length; i++)
    newA[i] = oldA[i];
  for (int i = oldA.length; i<newA.length; i++)
    newA[i] = EMPTY;
  return newA;
}
```

Update the draw code to use this now. We were already detecting if the array was full. Previously, in this case we just didn't add anything. Now, let's instead double the array so that we can always add the point. Make sure to double both the `x` and the `y` arrays.

```
  if (mousePressed)
  {
    int empty = nextEmpty(mousePointsX);
    if (empty == NOT_FOUND)
    {
      mousePointsX = doubleArray(mousePointsX);
      mousePointsY = doubleArray(mousePointsY);
      empty = nextEmpty(mousePointsX);
    }
    mousePointsX[empty] = mouseX;
    mousePointsY[empty] = mouseY;
    clr = (int)random(128)+128;
  }
```

Important! Notice how, after doubling the arrays, I called `nextEmpty` again? That's because we need to find an empty bin – `empty` was -1 so we need to find the next empty bin. Did we need to call the `nextEmpty` function again? There is a short cut to save work, do you see it?

19.7 **Searching Arrays: linear search**

In the previous example, we did quite a bit of searching on arrays. We had to search for an empty bin. We had to search for ellipses under the mouse. It turns out that searching is a very expensive operation. Imagine if we scale this up to 1 billion bins (not that unreasonable, e.g., in a high-end video game or searching on the web!). In this case, if we search for an empty bin, but there is no empty bin, we need 1 billion checks to confirm that. If the mouse doesn't touch any of the 1 billion ellipses, we need to check all 1 billion to be sure. That is slow!

There are a few ways to improve this. One way is to avoid searching – we can, for example, keep track of the next empty bin in the array cleverly so that we don't have to search for it. Another way is to be more clever with how we search.

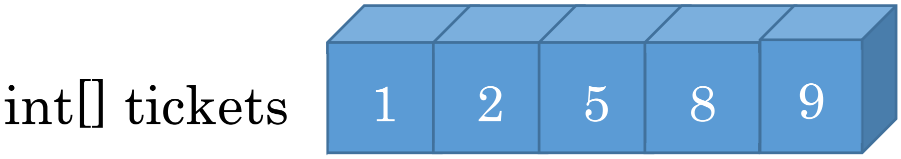
Searching is a fundamental problem of computer science, and if you stick with CS, you'll see it again and again. There are still great improvements being made to searching algorithms. Think about things like Google – Google is very fast at searching, it searches the entire web (!!!) in less than a second. This takes clever thinking, algorithms, and data structures (ways of storing data) to make it feasible.

Linear search – The simple, naïve approach to searching is called linear search. We just check every single element (like we did in the previous example). This is called linear because of you plot the search time against sample size (number of array bins, for example), you get a straight line. Twice the bins, twice the work.

You can imagine this as if you are searching a stack of randomly mixed exam papers. If I ask you to search a stack of papers to find the exam written by "Gumby", you have to search every one. If Gumby's exam isn't in the pile, you have to search *every single paper* to be 100% sure you didn't miss it.

Let's build an example to test out linear search. Let's make a list of one million lotto ticket holders. In this lottery, each ticket only has a single number, but they get large! We will start by generating 1 million ticket holders, each having a unique number. However, not all numbers are given out. That means that, when we draw a lotto number, we need to do a search to see if someone won or not.

One way to do this is to go through the array of people and hand out lotto numbers to them. We put a random space in between the numbers so that not all numbers are covered. For example:



Here, we have 5 people holding tickets, and each person has a unique number. However, as you can see, not all numbers are covered. There is no 3 or 4, 6 or 7.

We can make a function to hand out lotto numbers, and all we need is the array of people to put them into, and a global `RANDOM_SKIP` to determine how many we should skip at most. Be careful not to skip 0 as this would give us duplicate lotto numbers.

Also, it is useful to remember the largest ticket number so that we know what range of numbers we have lotto tickets in. Make the function return that number when it's done all the work.

```
final int RANDOM_SKIP = 3;

int handOutTickets(int[] people)
{
  int ticketNumber = 0;
  for (int i = 0; i < people.length; i++)
  {
    ticketNumber += (int)(random(RANDOM_SKIP)+1);
    people[i] = ticketNumber;
  }
  return ticketNumber;
}
```

This fills the array with random lotto ticket numbers. In `setup`, hand out those tickets, and `println` the array to be sure that it works as expected (no duplicates, etc.). I used a global array of tickets (1 per person), and a global `int` to keep track of the biggest ticket.

```
void setup()
{
  size(500, 500);
  biggestTicket = handOutTickets(tickets);
  println(tickets)
}
```

Once you confirm that this works, get rid of that `println`. A very large array can really wreak havoc on your computer if you try to print it.

Now that we have a data set, we can implement linear search. We can make a function called `linearSearch` which takes an array and an item, and returns the bin that the item is in. What should we do if it's not found? We can return -1 since it's an impossible value (make a final constant called `NOT_FOUND`). The algorithm is very simple:

- Assume that the item is *not* in the array – set the found index to `NOT_FOUND`
- Use a `for` loop to go through the entire array
  - if the data is found, then store the index
  - modify the `for` loop condition to quit early if the data is found
- `return` the index

```
int linearSearch(int[] data, int target)
{
  int index = NOT_FOUND;
  for (int i = 0; i < data.length && index == NOT_FOUND; i++)
  {
    if (data[i] == target)
      index = i;
  }
  return index;
}
```

Let's add a visualization to show which bins we actually looked in. We will draw a line across the screen to represent the range of tickets, with the smallest ticket on the left, and the largest on the right, and when an array bin is checked, we'll put a vertical line at the bin location to show that it's been checked.

First, use the following globals to define the line properties

```
final int CANVAS_SIZE = 500;
final int LINE_WIDTH = CANVAS_SIZE-100;
final int LINE_LEFT = CANVAS_SIZE/2 - LINE_WIDTH/2;
final int TICK_HEIGHT = 20;
```

and create two functions – one to draw the line itself at the specific `y` value:

```
void drawLine(int y)
{
  stroke(255);
  line(LINE_LEFT, y, LINE_WIDTH+LINE_LEFT, y);
}
```

And one to draw the tick mark. Since there will be more array bins than lines we need to scale it down. This function will need to know the bin number, and number of bins. This will also need to know the `y` value of the line.

```
void drawTick(int y, int bin, int binCount)
{
  float percent = bin/(float)(binCount-1);
  int x = LINE_LEFT + (int)(percent*LINE_WIDTH);
  line(x, y-TICK_HEIGHT/2, x, y+TICK_HEIGHT/2);
}
```

Finally, update your linear search to draw a tick mark at each bin checked. That is, inside the `for` loop, each time bin `i` is checked, draw a tick mark at bin `i`. (note: you will need to tell the function also the `y` coordinate of the visualization).

Now that we have these visualization tools, let's finish up the program. We already handed out the lotto tickets in the `setup`. Now let's pick a random number and check in the `draw` loop if anyone holds that ticket. We need a random ticket from `0..biggestTicket`, and then use linear search to check if there is a winner. It is also useful to toss out some text to give the info on the lotto draw:

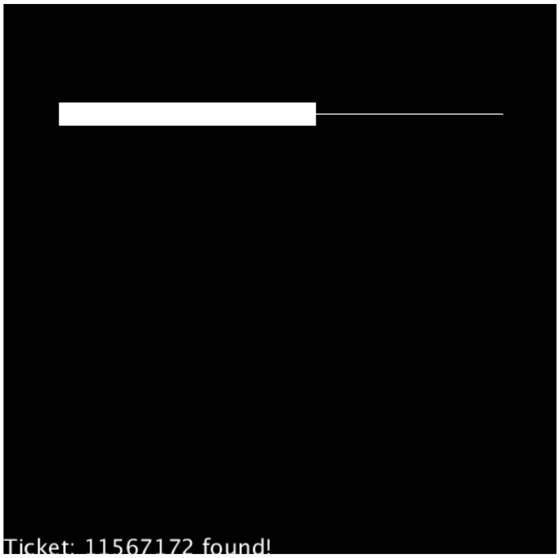
```
void draw()
{
  background(0);
  int ticket = (int)random(biggestTicket+1);
  drawLine(100);
  int result = linearSearch(tickets, ticket, 100);
  String s = "Ticket: "+ticket+" ";
  if (result == NOT_FOUND)
    s += "Not found";
  else
    s += "found!";
  textSize(20);
  text(s, 0, height-1);
}
```

Try it, it should work now. What you will see, is that each time a ticket is drawn, it will put tick marks on every bin checked, tell you who (if anyone) won, and then repeat. At only 100 tickets, this is very fast. However, upgrade to 1 million tickets and try again.

What you will see now, is a lot more easy to see as it's slower. Wow this is slow. Sometimes, you'll notice that a ticket is found very quickly, and in this case, the bar of tick marks will be very small (since only a few checks had to be made). However, if the ticket was later in the array, or not found at all, it can take a long time – several seconds – to find that out.

Upgrade to 10 million people, and it gets 10 times slower.

One way to speed this up is to recognize that the tickets are in a sorted order! As you go along, if you pass the ticket number that you wanted, you can stop searching. This will help in many cases, but it will still be very slow. However, it is this recognition that the array has an order that leads to a really powerful speedup.



```
Ticket: 11567172 found!
```

19.8 **Searching Arrays: binary search**

One of the fundamental approaches to searching is to make assumptions about our data. If we can have data with specific structure, then we can cleverly take advantage of that to save work.

For example, take a very large book, and search for page 237. Did you find it? How many pages were in the book, and, how many pages did you have to check? Did you have to check every single page? What if page 237 didn't exist (my kid tore it out!!). Do you have to check every page to be sure it doesn't exist? Of course not, you know that, in a book, the pages are in order, so you can be more clever. This is the essence of binary search.

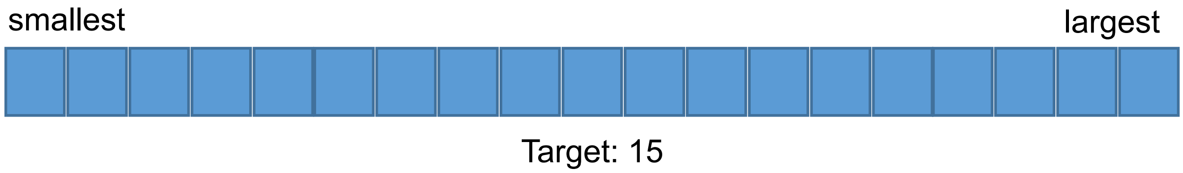
If we assume that an array of data is sorted, from smallest to largest, we can use binary search instead of linear search.

The key of how binary search makes things faster is that we end up ignoring whole sections of the array without checking them. If we can avoid checking them, we save work!

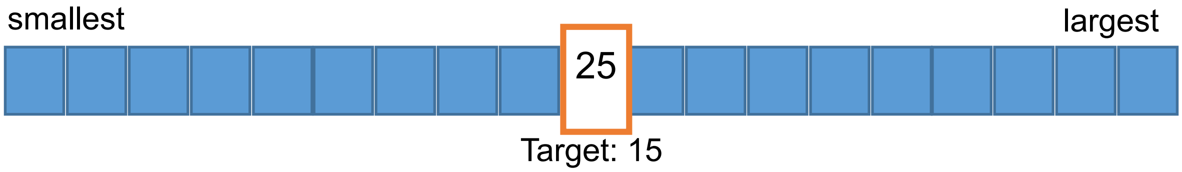
Here is the basic idea for binary search:

- We have an array of data, and a `target` we are searching for
- Assume the data in the array is sorted. If not, this won't work.
- Look at the array bin in the very middle of the array.
  - Is it our `target`? Yes? Woohoo! All done.
  - Is the data in the bin *smaller* than our `target`? Then let's immediately discount the left half of the array – no need to check those, as they are also smaller than our `target`.
  - Instead, is the data in the bin larger than our `target`? Then everything to the right is also larger, so we can discount those.
  - Repeat above. Calculate a new middle in the remaining bins

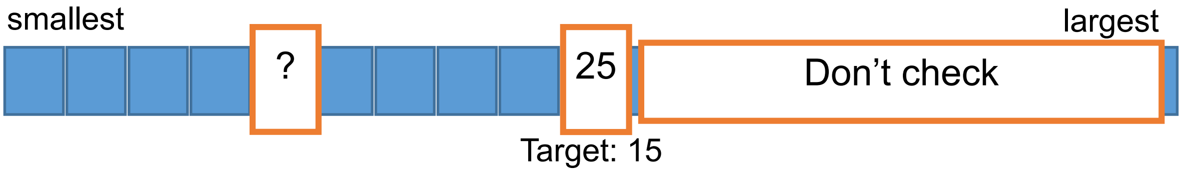
It is called binary search because, at each point, you make a binary decision – left or right. For example, here is an array, sorted. Say our `target` is 15.



First, we check the middle. The data happens to be 25:



So, since the array is sorted, we know that everything to right of 25, is either 25 or bigger. So all of those are bigger than 15, and we don't need to check them. Immediately we throw array half the away. We then repeat with a new middle of the smaller array:



Depending on this check, we'll throw away the left and right, halving our array again.

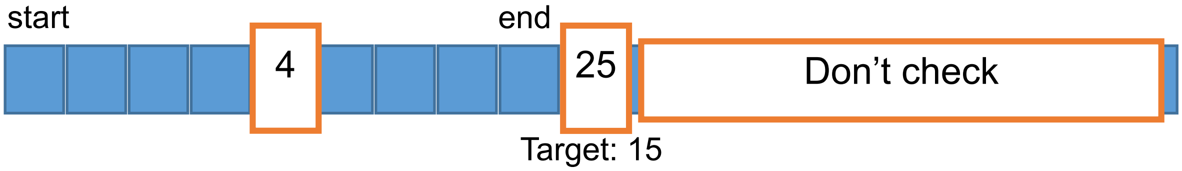
Here is the specific algorithm:

- Repeat:
  - Set the `start` and `end` of your array. `start = 0`, `end` is `length-1`.
  - `midpoint` is `(start + end)/2`
  - Check `midpoint` against target
    - If `data > target`, cut right half
      - `end = midpoint - 1`
    - if `data < target`, cut left half
      - `start = midpoint + 1`
    - if `data = target`, we're done!

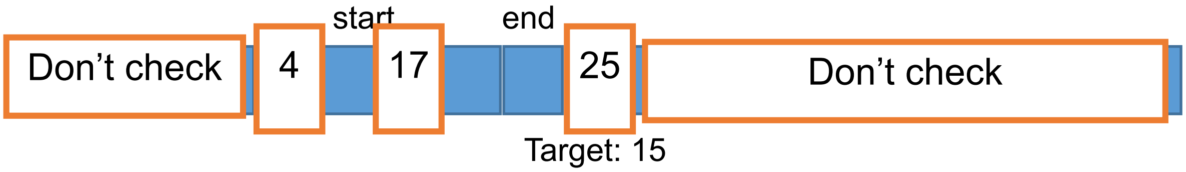
Notice that we always add or subtract 1 from the mid-point. This is important – we already checked the midpoint, so we should completely exclude it from the array.

When do we stop? We have two possibilities: a) we found the target, or b) there is no target. Knowing when we have a target is easy, but when do we know that there is no target?

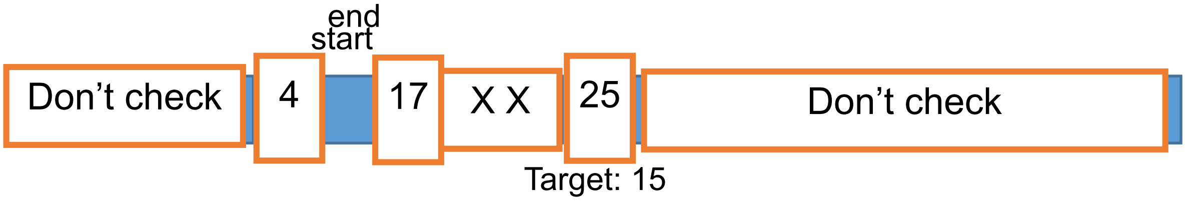
It happens like this. Eventually, `start` becomes bigger than `end`, which is illogical (`start` should always be smaller!). This happens because eventually the midpoint becomes `end` or `start` (or both!), and when we place the `start` or `end` past the midpoint, they leap frog. See, if we continue the previous example, moving start and end as we go:



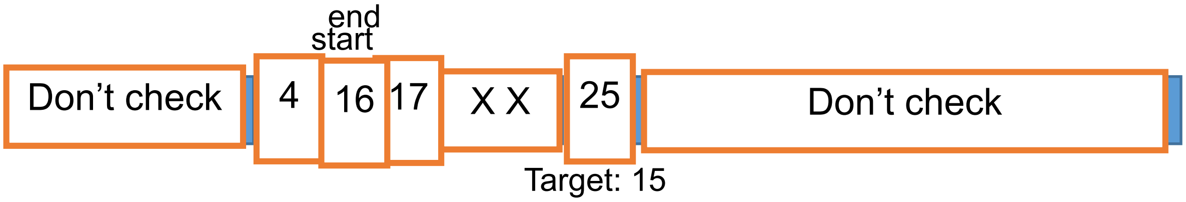
Since `midpoint` data was 4, which is smaller than the `target`, we discount the left, and move `start` one past the `midpoint`, and have a new `midpoint`



17 is bigger than `target`, so we move the `end` in. Now `end` and `start` are the same and we only have 1 bin left.



This is a 16, so we move the `end` to `midpoint - 1`, which is before start. Since `end < start`, we are done!



Notice how in an array of 19 elements, we only had to do 4 checks to determine that the item wasn't in the array! That is a lot less than linear search, where we would have had to check all 19.

Now, let’s implement this in processing code, making a function called `binarySearch` that takes the array to search, the `target`, and also the `y` coordinate for plotting our progress.

First, we need the `start` and `end` variables, set to the first and last bins. We also need our `index`, assuming that our target is not found:

```
int start = 0;
int end = data.length-1;
int index = NOT_FOUND;
```

Since we don’t know how may checks we need, we should use a `while` loop. We loop while `start` and `end` do not overlap, e.g., while `start <= end`, AND, while we haven’t found the target yet. If either of these become true we should quit.

Each time through the loop, we calculate the `midpoint`, and look in the array at the `midpoint`. If the data is larger than the `target`, then we move the `end` point. If the data is smaller, we move the `start` point. Otherwise, we have the data and we should save the index!

Don’t forget to draw the tick marks at each bin looked at as in the previous example:

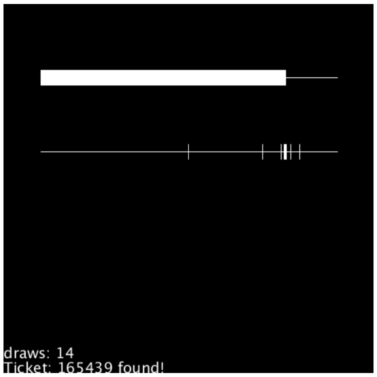
```
int binarySearch(int[] data, int target, int y)
{
  int start = 0;
  int end = data.length-1;
  int index = NOT_FOUND;
  while (start <= end && index == NOT_FOUND)
  {
    int midpoint = (start+end)/2;
    if (data[midpoint] > target)
      end = midpoint-1;
    else if (data[midpoint] < target)
      start = midpoint+1;
    else // data == target!
    index = midpoint;
    drawTick(y, midpoint, data.length);
  }
  return index;
}
```

That’s it! Now we can use this instead of linear search in our `draw` loop, to see how fast it is. All that you need to do is to change the function call in `draw`. Try it starting with the 10,000,000 tickets we tried last time that really slowed down linear search.

How large can you make the search array and still keep it fast? Be careful of ticket number overflow.

Play with this. Try to add some Booleans to make it pause so you can see the result. Try comparing against linear search by plotting the binary search on a different line and running both in parallel.

You should see a significant time savings. Linear search becomes unreasonable soon after 10 million cases, but binary search should stay fast up until the maximum array size you can make. This is the power of algorithms, and the power of Computer Science!



```
draws: 14
Ticket: 165439 found!
```

19.9 **Searching Arrays: complexity**

It is very clear that binary search is faster than linear search. But how much faster is it? Doesn't it depend on the size of the dataset? Are there algorithms faster than binary search?

It turns out that in Computer Science, we have standardized measures and techniques for explaining these differences. If you stick with the program, there are whole courses on analyzing and comparing different algorithms. This may just seem mathy, but it is actually very important, as practicing computer scientists need to have the skills and knowledge at their fingertips to select and develop appropriate algorithms for your purposes.

One way to specify how fast an algorithm is, is to count the number of operations you need to perform in terms of `n` (the search space). We often look at best case, worst case, and expected case. For example, with linear search, the best case is 1 check – the first element is what we're looking for! The worst case is `n` checks – the item doesn't exist. It turns out that the expected time (average) is about `n/2`

With binary search, the best case is still 1 check (the target is right in the middle!). The worst case, however, is about $\log_2(n)$ (log base 2 n) – I'll leave it for a later class to learn how to calculate that. The expected average time is $\log_2(n)-1$ – just one check fewer than the worst case. The following table gives a breakdown:

algorithm	best case	worst case	average	10 elements	1000	1 bil.	1 septillion (24 0s)
linear	1 check	n	n/2	5	500	500mil	.5 septillion
binary	1 check	$\log_2(n)$	$\log_2(n)-1$	3	9	29	79

Wow! That's quite the difference! Linear search quickly explodes, while even at a septillion items to check, binary search only takes 79 checks.

Check your Understanding

19.10 Check Your Understanding: Exercises

Exercise 1. Use the count-as-you go partially-filled arrays technique to implement a kind of *undo* function in a drawing program. First, start with a simple drawing program as shown in the examples, using the count-as-you-go partially-filled arrays technique.

a. If a key is pressed in the keyboard, remove the *last* entry entered, so that it undoes the last drawing command.

Exercise 2. Use the impossible-value partially-filled arrays technique to implement a program that has random balls popping up on the screen. When a key is pressed, balls get erased from the left side of the screen, kind of like sweeping.

a. Make a program where, each draw, an ellipse is added and drawn at a random location. Use a partially filled array to store the ellipse location
b. If the array is full, grow it to double the size
c. If a key is pressed, erase the left-most ellipse. You can do this by finding the smallest `x` value, and erasing that ellipse. There may be several ellipses with the same value, just pick one. While the key is pressed, draw a horizontal like to simulate a broom sweep.

Exercise 3. Update Exercise 2 to use a parallel, boolean array, instead of an impossible value, to mark empty bins.

Exercise 4. Go back to Section 14.10, where you implemented a space shooter with a bad guy on the screen. Upgrade that example to have 10 bad guys, using the impossible-value partially-filled arrays technique.

Exercise 5. If binary search is much better than linear search, then why would you ever use linear search at all?

Exercise 6. The Sieve of Eratosthenes is a famous, classic method for finding prime numbers (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). Implement the sieve to find all prime numbers up to 1,000,000. The idea is really simple, but implementing it can be a little tricky. The hard parts are that you need multiple, nested loops, and, you always need to be careful not to run off the edge of the array. Here is the algorithm

> First, generate a list of numbers 1..n, where n is your maximum number (1 million, in this case). Then, starting with the number 2, *cross off* all multiples of 2 in the list. In programming, start by making an array of size `n` that has the numbers `1..n` stored in it. Then, for all `2p` while `2p < n`, (start at p or 2, 3, then up, etc.) set the array bin of `2p` to be empty.
>
> This removes all multiples of 2 from the array, as they are clearly not prime numbers. Next, starting at 2, look in the array and find the next non-empty bin. In this case, it will be 3. Repeat the above process for removing all multiples of 3. Then, again find the next non-empty bin. 4 will be empty because it is a multiple of 2, but 5 will be in the array, so remove all multiples of 5. Be careful not to run off the edge of the array.
>
> You stop when your search for the next non-empty bin runs off the edge of the array.
>
> You will need several loops. Use while loops as it is non-trivial to calculate how many times you need to run the loop. Your main loop will run while you are still finding numbers to cross off multiples of, that is, while your next number to work on is less than the size of the array. Inside that loop, you need
>
> - A `while` loop to go through the multiples of your current number, and cross them off. Start with the number 2: cross off `2p` for all `p` while `2p` is less than your maximum.
> - Once that's done, a while loop to find the next number to cross off multiples of. E.g., you will find 3 the next time.
> - Loop to the first step, removing all multiples of 3, then 5, etc.
> - Once you are finished, the remaining numbers in the array are prime numbers. Make sure to exclude 0 if your solution used 0. Use a for loop to print out all the bins with numbers, they will be primes. Maybe start with a small size (100) first, as you can easily verify that they are primes.

Exercise 7. Make a drawing program that uses the count-as-you-go method for partially filled arrays, that lets the user draw a path, storing the points in two arrays, one for `x` and one for `y`. Make the arrays store up to 1000 points. In `draw`, draw lines through the points in order. You need a variable to remember where the next empty bin is. Be careful to check the end of the array.

a. If a key is pressed on the keyboard, delete every second point – do this by halving the size of the arrays and only keeping every second entry

b. Alternatively, if a key is pressed, double the size of the array, with new points being inserted in the middle of the existing two, as the average of both. For example, given `{1,2,3}` you would get `{1, 1.5, 2, 2.5, 3}`.

Exercise 8. Make a drawing program that lets the user draw paths with the mouse, remembering the current path and the previous one drawn. They are drawn using different colors.

a. It has 2 sets of `x,y` arrays to store series' of points. One called `current`, one called `previous`
b. When the mouse is pressed, the mouse's location is added to the `current` array until the mouse button is released. If the `current` array gets full, stop adding.
c. The next time the mouse is pressed, the `current` points become the `previous` points, and we start with a fresh, clean `current`. You do not need to do any copy operation, just clever work with the variables.
d. The current path is drawn in 255 white, and the previous path is drawn in 128 grey.
e. Update the program so that when `current` gets full, it automatically

Exercise 9. A limitation of binary search is that it only works in one dimension as we have learned it. What would it look like in two dimensions (e.g., pixels on a screen), three dimensions (objects in 3D space), or so forth? Look up, on Wikipedia, Quadtree and Octree.

Exercise 10. Earlier in the course, we used nested `for` loops to draw a tic-tac-toe board. You now have all the pieces you need to implement a game. Try implementing a basic version of this game, using partially-filled arrays to mark a game square as either empty or having data. The logic to detect a winner is a little tricky!

Exercise 11. Do exercise 7 but instead with the impossible-value strategy for partially filled arrays. Part b in particular is quite tricky as you have to average a bin's neighbors.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

UNIT 20. END OF THE COURSE

Welcome to the end of the course!

What did you learn in this course? You may think that you learned Processing, but actually, you learned fundamental programming techniques. If you stick with computers, regardless of the language you us you will most likely see integers, floating points, functions, booleans and ifs, loops and arrays, in your next language. More importantly, the thought processes you have been applying to use these tools to solve logic transcend computer science.

For example, here is a program in a language that you have most likely never seen before. This is a real program, can you guess what it does? Can you quickly figure out what some of the constructs are?

```
var
  endPoint, startPoint, midPoint, target,
  indexFound : Integer;
  data : Array[0..11] of Integer
       = (1,1,1,1,2,4,5,5,10,90,100,1000);
Begin
 startPoint := 0;
 endPoint := Length(data)-1;
 target := 101;
 indexFound := -1;
 while (startPoint <= endPoint) AND (indexFound = -1) do
 begin
   midPoint := Floor((startPoint + endPoint) /2);
   if data[midPoint] = target then
     indexFound := midpoint
   else if data[midPoint] < target then
     startPoint := midPoint+1
   else
     endPoint := midPoint -1;
 end;
 writeln(indexFound);
End.
```

This is a program written in Pascal, a language used for teaching. Things are a little different, but should be reasonably comfortable. See? You learned fundamental programming.

Because of this, as you advance, many text books will not give examples in any particular language. One common method is to use *pseudo-code*, readable descriptions of algorithms that don't match any particular language. For example, here is some pseudocode you may see in a text book:

```
for j ←1 to length(A)-1
    key ← A[ j ]
    i ← j - 1
    while i >= 0 and A [ i ] > key
        A[ i +1 ] ← A[ i ]
        i ← i -1
    A [i +1] ← key
```

You may not understand exactly the nuances of what the outcome is, until you work through it, but the step by step pieces should make reasonable sense.

You are done with the course! As a bonus, this book contains two extra chapters that are not part of the course itself. As such, these chapters do not have the exercises or formatting of the previous chapters. First, you can get a quick introduction to Object Oriented Programming, and second, you can see how to move from Processing to raw Java.

You're done! Any feedback, suggestions, corrections, etc., are welcome: young@ca.umanitoba.ca

UNIT 21. INTRODUCTION TO OBJECTS

Objects are not an official part of this course, and this section is not testable. However! Objects are awesome and so I thought it would be great to put them in the notes.

Objects are the next natural progression in this material. So far, we learn a new technique, find the limitations, and learn another new technique to overcome those limitations. Our latest limitation is that it is very hard to pass around a lot of data to and from functions. Also, you end up getting larger and larger messes of variables for components of your program, which becomes unmanageable as your program grows. For example, let's make a bad guy that flies around a screen. The bad guy has an `x` value, a `y` value, a width, a height, a color, and a move speed.

```
int badGuyX;
int badGuyY;
int badGuyW;
int badGuyH;
int badGuyClr;
int badGuyMaxSpeed;
```

We have learned how to scale this up to 1000 bad guys, using arrays. That is easy. There are remaining problems, however. Imagine a function to draw this bad guy. It may look like this:

```
void drawBadGuy(int x, int y, int w, int h, int clr,
    int maxSpeed);
```

this is tedious, but doable. Every time we call this we have to type in all of those bad guy properties. It gets worse if we have more properties (rotate speed? Bullets left? Health? Shape? 10 others?).

Another problem is that we can only return one value from functions, and we cannot change those values in a function. So the following function simply is not possible.

```
void moveBadGuy(int x, int y, int maxSpeed);
```

We can use max speed to move the bad guy, but we cannot get the modified x and y out of the function. It is lost, because only the local copies in the function are modified.

The solution to this is objects. Object oriented programming is a huge topic, and there are many advanced avenues and dusty corners – you will learn about it in great detail in a Computer Science degree. However, basic objects are not so bad, and you have learned a lot of the skills already needed to make them.

Basically, objects are a nice way to collect data in a clean wrapper. You can “chunk” data together. For example, a `BadGuy` object would have a bunch of properties, such as an `x`, `y`, `w`, `h`, `clr`, `maxSpeed`, and many others. We setup an object to have all this in one package, and then we can use it like other types. We can pass an object to a function, and pass an object back, and everything stays grouped together.

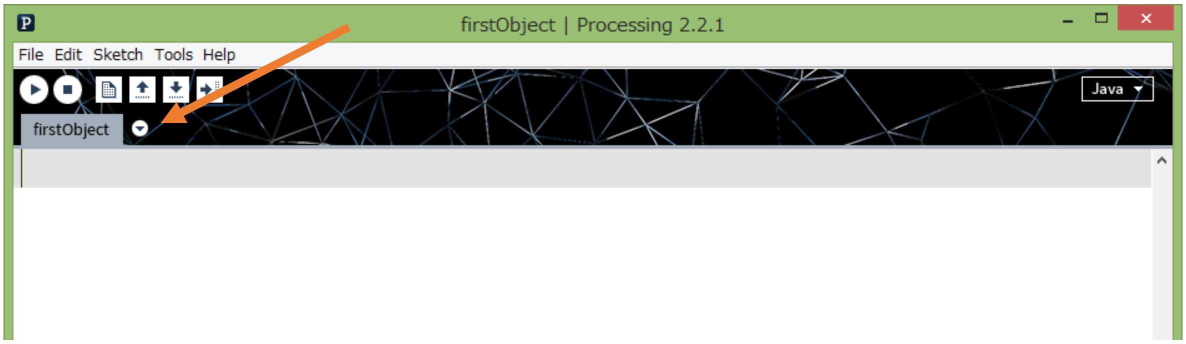
21.1 **Creating Your First Class**

We have many data types in processing. We have the primitive types (`int`, `float`, etc.), and the array types. We also have `String`, which we mentioned before, is an object.

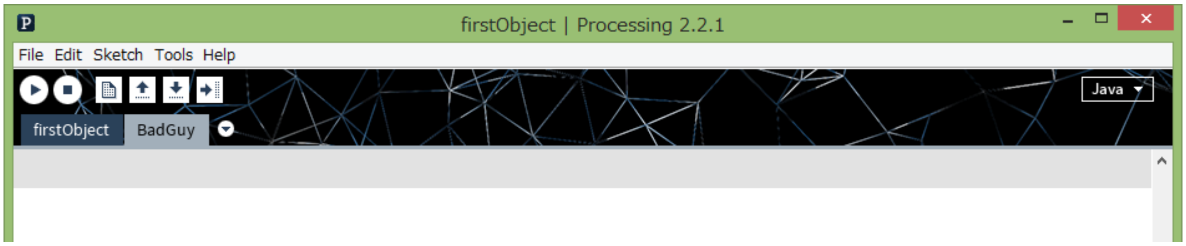
To create our own objects, that work a little like the String type, we need to define our own, new, data type. This is called a `class`. Then, once we create this new type, we can create new variables that use this type. Finally, there is a new step for actually making the object work.

Perhaps the best way to explain this is to show it.

First, in processing, you need to add a new tab, which adds a new file, to your project. Click on the down arrow next to your project name:



And choose “new tab”. It will ask for the file name. Type in `BadGuy`. A new tab appears that has no code in it.



Select this tab, and we can type more processing code in here.

To create our own object type, we need to learn new special syntax to make our first class. The syntax of a class is very simple:

```
class ClassName
{
} // end class
```

We can specify a class name, and now our program has a whole new type that we can use. Convention has classes start with a capital letter (like `String`!) and it is good practice to call the class the same as the tab / file name (this is required in pure Java). We will call our class `BadGuy`:

```
class BadGuy
{
}
```

So far, this is not very exciting. We have created our own class, but we did not put anything into it. It turns out that we can put stuff inside the class, and it all gets grouped together under the `BadGuy` name. For one, we can add variables (which is where this example started). We can also add other things, like functions (called methods when they are part of an Object), but that is for a later course.

Let's add a collection of variables that define the bad guy.

```
class BadGuy
{
  int x;
  int y;
  int w;
  int h;
  int clr;
  int moveSpeed;
}
```

We can make any variables in here we want, just like in our main program. Arrays, too!

Now, **be careful**: You cannot store data in these (yet). At this point, you are just defining your brand new data type. A class just specifies what will be in an object if you create one, what kinds of data can be stored. This is just a blue print, a template. We need to learn how to make objects first!

21.2 **Instantiating Your First Objects**

Classes are a new data type that we created. We cannot actually store data in a type, we need to make variables with that type.

Go back to your main tab. Now that we created a class called `BadGuy`, we can use this like any other data type in our program – just like `String`, for example. Let's make three global variables for the bad guys:

```
BadGuy b1;
BadGuy b2;
BadGuy b3;
```

So making variables using your new class is easy. However, we need to learn some new things before you can use them. Just like arrays, you need to *instantiate* your object before you can use it. Also, these variables, just like arrays, only store the address of where the object is in memory – this has all the implications that arrays have, e.g., for use with functions, and the == operation.

- You need to *instantiate* an object before you can use it, similar to how you must instantiate an array.
- The object variable only stores a memory address of where the object is located in memory.

To instantiate an object, that is, to create a new object, we need to use the `new` keyword. There is a small difference from arrays: you put () after, like a function call:

```
ClassType variable = new ClassType();
```

For example:

```
BadGuy b1 = new BadGuy();
```

The `new` command goes off to memory (just like with arrays), allocates enough memory to store those variables you defined in `BadGuy`, and comes back with the memory address. The memory address is then stored in the variable for use later. Let's do it for all three:

```
BadGuy b1 = new BadGuy();
BadGuy b2 = new BadGuy();
BadGuy b3 = new BadGuy();
```

Just like with arrays, when we call `new` three times, we get three different spots in memory. We have three bad guys, all with their own copy of the variables inside of them. They are stored at different locations, and changing one does not change the other, they are separate!

Here is a rundown of the terminology we use:

class – the blueprint of the data to be stored and functionality to be included

object – an actual instance of a class in memory that can be used to store data, as described in the blueprint

instance – see object

Note: sometimes, there are nuanced differences between the term instance and object, depending on the language. For most people they are the same.

21.3 **Accessing Instance Variables inside Objects**

Now that we have created the three objects, we need to learn how to access the object's variables (called *instance variables* since there is one set per instance of the object). You do this by putting a dot after the variable name, and then following with the variable name.

```
object.instanceVariable
```

for example

```
b1.x
```

What happens here, is that Processing looks inside `b1`,

this now acts just like any other variable. You can add to it, subtract from it, use it in a formula, etc.

For example, we can setup one of our bad guys in the setup block:

```
void setup()
{
  size(500,500);
  b1.x = 10;
  b1.y = 10;
  b1.w = 100;
  b1.h = 50;
  b1.clr = 255;
  b1.moveSpeed = 10;
}
```

Remember: here we don't use the `new` keyword because the object has already been created (*instantiated*) globally. Just like with arrays, objects always need to be instantiated.

We can use this to move, and then draw, the bad guy, in the draw block. The same as above, we use these like any other variable:

```
void draw()
{
  background(0);

  // move bad guy
  int move = (int)random(b1.moveSpeed*2)-b1.moveSpeed;
  b1.x += move;
  move = (int)random(b1.moveSpeed*2)-b1.moveSpeed;
  b1.y += move;

  // draw bad guy
  rect(b1.x, b1.y, b1.w, b1.h);
}
```

So this should work, but so far we are not really saving much work. I could do the above example without objects, and just make my own variables; I would even save typing while at it!

21.4 **Objects and functions**

The power of Objects starts to become obvious when we use them with functions. Classes are just another type, so we can toss objects back and forth from functions no problem. Further, because objects work like arrays – we toss the memory address around and not all the data – functions can actually modify the object.

Let's make a new function, that generates a new `BadGuy` and sets it to some random values. It doesn't take any parameters, but will return the new bad guy

- Create a new object instance of `BadGuy`
- Set the instance variables to reasonable but random values
- Return a reference to the object (the memory address). Make sure to set the return value of your function to the Class type as below

Try it out. This is what I came up with

```
BadGuy newRandomBadGuy()
{
  BadGuy b = new BadGuy();
  b.x = (int)(random(width));
  b.y = (int)(random(height));
  b.w = (int)(random(MAX_WIDTH)); // 100
  b.h = (int)(random(MAX_HEIGHT)); // 100
  b.clr = (int)(random(255));
  b.moveSpeed = (int)(random(MAX_SPEED)); // 10
  return b;
}
```

Now we can use this to create as many bad guys as we want. Since we instantiate the object in this function, we don't need to do it at the top of the program any longer. Also, update your setup to call this function and set your `b1`, `b2`, `b3` to their own random bad guys.

```
void setup()
{
  size(500,500);
  b1 = newRandomBadGuy();
  b2 = newRandomBadGuy();
  b3 = newRandomBadGuy();
}
```

We now have three random bad guys! Let's make two more functions. First, let's draw a bad guy:

```
void drawBadGuy(BadGuy b)
{
  stroke(b.clr);
  fill(b.clr);
  rect(b.x, b.y, b.w, b.h);
}
```

This takes in a `BadGuy` variable, and uses the instance variables to draw as needed. Next, let's make a function to move a `BadGuy` around. This is new! This was not possible before, without objects. It should take a `BadGuy` variable, modify it in place, and not return anything.

```
void moveBadGuy(BadGuy b)
{
  int move = (int)random(b.moveSpeed*2)-b.moveSpeed;
  b.x += move;
  move = (int)random(b.moveSpeed*2)-b.moveSpeed;
  b.y += move;
}
```

This function takes a reference to a bad guy (a memory address), and uses that to get the instance variable values, AND, change them. Changes made here are permanent inside the object that was passed to us.

All we need now is to update our draw block:

```
void draw()
{
  background(0);
  moveBadGuy(b1);
  moveBadGuy(b2);
  moveBadGuy(b3);
  drawBadGuy(b1);
  drawBadGuy(b2);
  drawBadGuy(b3);
}
```

This is very powerful. Combine this with arrays, and – wow! We can do a lot. Keep at CS and you will learn even more powerful tools.

UNIT 22. INTRODUCTION TO JAVA

As I said earlier, Processing is basically Java. However, to work in real Java, you will need to let go of the graphics for now, and, learn about all that syntax that Processing kept hidden from you. In addition, you need to have a shift in thinking about how you approach your problems.

Your work in processing followed a very fixed model: you have a draw loop, and some global state, and each frame, you update your state. The update can be based on calculations (e.g., a falling ball), input (a mouse click), or even random values. Although you have seen some examples that either use static processing, or the `noLoop();` command, to make the program run only once, this has been the exception.

In Java, like in most programming languages, things work a little differently. ***When you start your program, your code only runs one time, and then finishes.*** There is no draw loop, there is no repetition. If you want to have an animation or some kind of event loop that happens, e.g., on a timer (like Processing, 60 times a second) or input (mouse press). This is a lot like the static programs you made, except that they are fully functional, with functions, arrays, etc.

This may look to be a simple distinction, but as a result of this, many of your programs will look quite different than your programs to date. In Processing, most of your work has been in state management. From time to time, you have done data pre-processing, but that was less of your work. State-management programming is only one paradigm and one kind of problem to solve; the world of programming is much bigger.

One common model of programming is the input-process-output model. Your program gets some data (e.g., from a file, from the user), and stores that data somehow in variables. Then, once input has been all received, you process the data to get some result – e.g., you may average the data, check validity, etc. We provide example of such problems in this chapter.

It is strongly recommended to complete the exercises in this chapter in preparation for your next programming course in pure Java. This will not only refresh your memory on the basics learned in this course, but will give you some experience in pure Java that you will need to succeed as you continue.

22.1 Moving to Java

There are two very important differences that you need to address when moving to Java:

- Processing gives you a free and easy-to-use graphics library. Java doesn’t do this (it takes a lot more work).
- Processing is a wrapper around Java that simplified the syntax.

Luckily, the non-simplified syntax it isn't so bad for the most part. Much of the syntax is related to object oriented programming. Even if some of the keywords do not make sense to you yet, you will learn them in your first course on objects, very quickly.

To get Java working, you need:

- A Java Compiler
- An editor that will run the Java tools for you
- To learn the new Java syntax

You should use the official Java Compiler, from Sun. This is called the JDK – Java Development Kit – and it includes all the software needed to compile and run your Java programs. Before we jump in, however, let's talk a little more about the changes of moving to raw Java.

Getting the Java Development Kit. The language and terminology surrounding Java is very confusing – those people love acronyms. You will want to get the Java SE version (Standard Edition). Make sure to get the newest version of it. Instead of putting a link here, your best bet is to go to google and type in Java SE JDK.

Don't get the JRE – that only lets you run Java programs, not compile and make your own.

If you are having trouble getting this to work, try uninstalling all Java versions that may be on your computer already. Java has a lot of versions and they have a way of getting in the way of each other.

Java Editor. There is no "standard" Java editor, for example, like Visual Studio for C#. There are many that you may have heard of, as shown here. You can find these below using google.

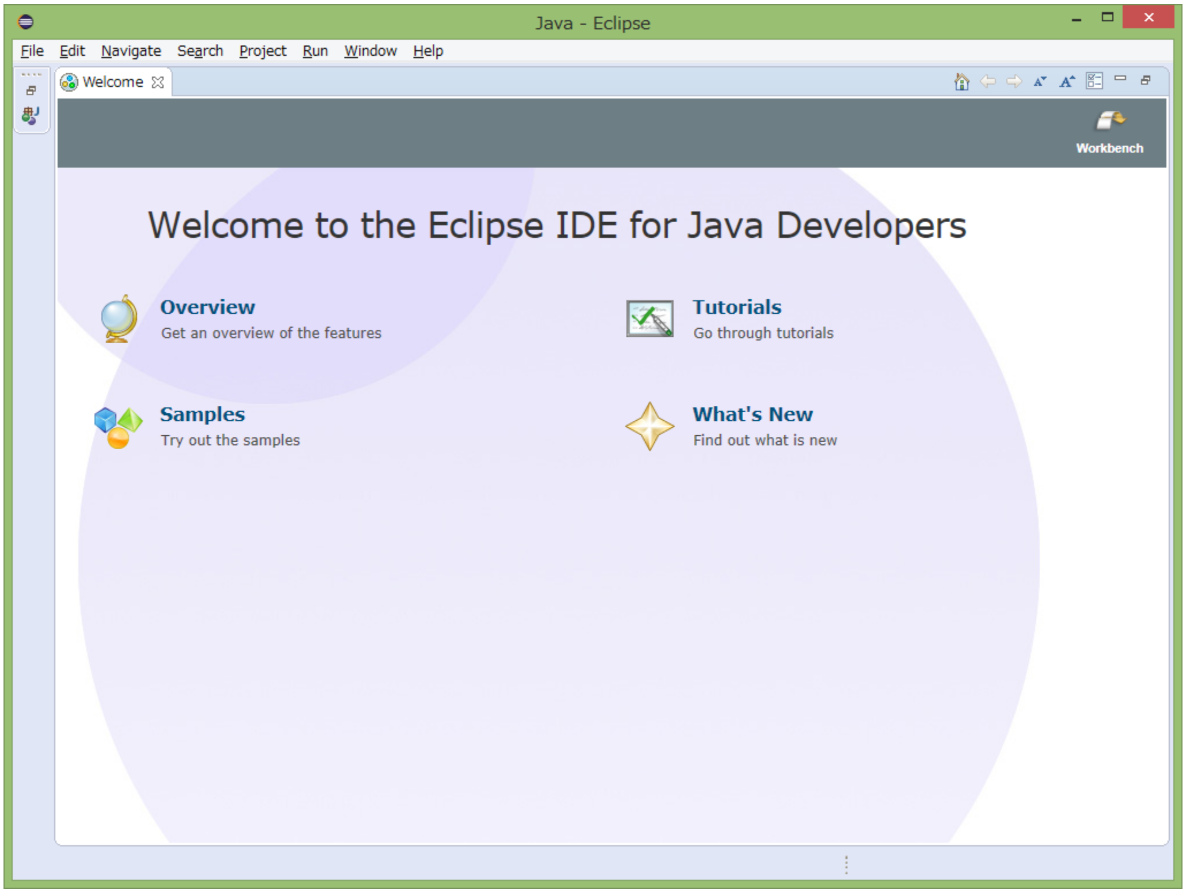
- **Dr Java** is a teaching tool, a Java editor for learners and beginners.
- **Textpad** is a similar tool, but for windows only.
- **Eclipse** is a professional tool for Java development.
- **BlueJ** is the official editor of Green Foot, which is another graphics system for Java
- **Sublime** is a popular IDE that runs on many platforms.

I personally recommend Eclipse – it may be like jumping off the deep end of a pool, but once you learn it, there is a great deal of power in that editor.

To get started, download eclipse, and un-zip it into a target folder. Ask someone to help if you are less familiar with this kind of task.

When you start Eclipse, it will ask you which workspace to use. You probably don't have a workspace setup, so just select the default. (Alternatively, you can save on your DropBox or other backup tool, although there are a lot of files it creates that makes backups busy)

When you first start Eclipse, and choose your workspace, you will see a page along the following lines:



Welcome to the Eclipse IDE for Java Developers

Overview
Get an overview of the features

Tutorials
Go through tutorials

Samples
Try out the samples

What's New
Find out what is new

To get started, click the “Workbench” icon on the top right.

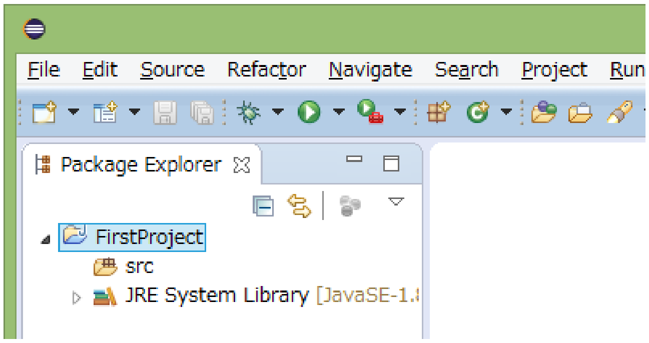
Then, you want to create a new project. It is a good idea to save each assignment, lab, etc., as its own project. Inside one project, files can conflict with each other.

To do this, go `File->New->Java Project` (if you only have Project and not Java Project, select Project and inside that window, select Java. If there is no Java, you may have downloaded the wrong Eclipse version).

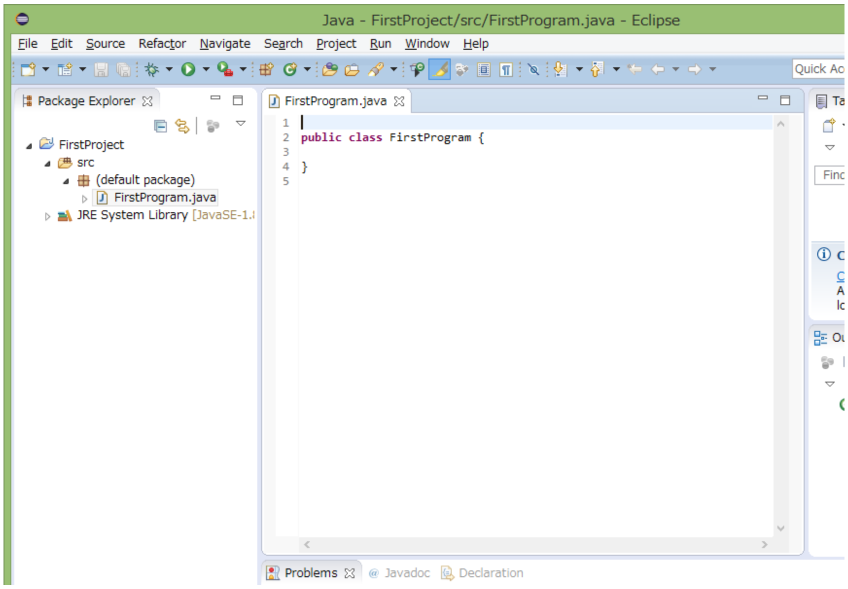
A pop up window comes up with all kinds of settings. All that you need to do is to create a project name.

Type `FirstProject` in the project name and click “`Finish`”

You will see your project on the left, in the “package explorer” section. Click the down arrow and you will see that it is linked to the JRE System Library (basic Java), and you have a folder called `src`. This is where you put your source files. That is, your programs that you type up.



Right click on the `src` folder, click “`new`” and click “`class`”. Again, you are presented with a whole slew of options. All that you need to do is to give it a name “`FirstClass`”, and click finish. It will warn you not to use the default package, but ignore that for now.



```
public class FirstProgram {

}
```

Now you have a source file open, ready to take in your first Java program!

The above two steps – getting Java installed, and getting an editor, is perhaps your biggest hurdle in the transition. Once the software is setup, it gets easier.

Once you type programs, you can compile and run them by pressing the “run” button, the green and white triangle on the toolbar.

22.2 **Basic Java Syntax**

The basic Java syntax is a little different than what you may be used to. Unlike processing, where you can just start typing a program, all Java code requires basic object-oriented syntax, even if you’re not using it. That is, ***all code must be in a class*** (see the previous unit regarding what a class is). When you create your first program, you do so by creating a class wrapper (syntax below), and place all your code inside of that.

Second, all Java programs start at a function called `main`. In Processing, your program started at `setup` and then wend to `draw`. Here, it’s simpler: when you run your Java program, the `main` function runs one time, and then the program quits.

Below I provide a template with all the above components in place. There is some syntax that looks confusing, but you will learn about it soon. For now, `public` is an access modifier – it specifies which parts of the program can access those components. `static` means that the code should work even without instantiating an object. The `void main` takes some parameters since the program that starts your program can send some strings in, an array of strings, in fact.

```
public class FirstProgram {
      public static void main(String[] args)
      {
      }
}
```

Important: Your class name `FirstProgram` MUST MATCH your file name “`FirstProgram.java`”. This is a Java standard, and it won’t work if you don’t do this.

If you run this program, Java will run the `main` function, and end the program. In this case, absolutely nothing happens, because your `main` is currently empty. Again: unlike in processing, Java does not have a draw loop that gets called continuously – if you want that, you will have to learn how to build it yourself. All that Java does is to call your main function once, and stop.

Except for very few exceptions, you cannot put code outside the class. You can make variables and functions inside the class, like you are familiar with; we will see examples of this soon.

Java is strongly typed, just like Processing, which means that every expression and variable has to have a specific type, and, it’s a pain to go between types.

One key difference between Processing and Java is that, for floating point numbers, Java defaults to `double`, while Processing defaults to `float`. For example, the following would work fine in Processing but does not work in Java:

```
float f = 1.23;
```

This does not work because `1.23` defaults to the `double` type, and you cannot store a `double` in a `float`. You can fix this with a cast:

```
float f = (float)1.23;
```

Other than that, the Java types should work as you have seen in Processing.

22.3 **Standard Commands**

The above program is pretty useless, as there are no commands in the main. We need to learn some commands in pure Java.

Unlike in many languages, in Java you often fully describe where in the system's command hierarchy (actually, object hierarchy) a command is located, in order to use it. Translation: it's ugly and messy. This means a lot of memorization, as many commands require this full description.

Perhaps the most basic command is tossing text out to the console. You do this with:

```
System.out.println(<some variable or text>);
```

This command is almost identical to the Processing version, except you tell the computer first to look in the `System` library, then in the `out` section, and use the `println` command from there. Toss the following command into your `main` function:

```
System.out.println("hello there");
```

And run your program (`Run->Run`). You should see the message "hello there" pop out at the terminal or console in your editor. Make sure to find this, as it can be hard to find, but the console is very useful and very important.

This `println` command is not exactly like the Processing version, as Processing has a few features. For one, if you try to print an array, Java just gives you the memory address stored in the variable (mostly useless) – if you want to print the array contents, you need to do it yourself with a loop.

You also have the `Math` commands, which include the familiar commands `max`, `min`, `sin`, `random`, and so forth. You use these by prefacing them with `Math.`:

```
int i = Math.max(10,20);
System.out.println(i);
```

Java has a pretty handy (but hard to navigate) reference manual online. For example, the following explains all the functions in the Math library:

`http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html`

In general, you can save time by being familiar with this reference to lookup the specifics of the tools you will use.

22.4 **Libraries**

In addition to built-in Java commands, Java comes with an extensive library of additional code that you can use. However, if your program wants to use additional libraries, you need to tell Java to link to that library. It does not do this by default for efficiency reasons.

To link to a library, you need to use the `import` command. This goes at the very top of your program, outside the class definition:

```
import <library name>;
```

For example (we'll use this in a minute): The asterisk symbol (`*`) is commonly used in computer science to mean "all".

```
import java.util.*; // import whole util library
```

22.5 **Getting Basic User Input**

Unlike in Processing, getting user input in Java is not nearly as nice. Forget about the mouse for now, or detecting individual key presses. Let's start by getting a full string from the user.

To do this, we need to use a new class called `Scanner`. `Scanner` connects to input sources (like the keyboard) and gives us some commands to get text from that source. Scanner is an object, so we need to use special syntax to get it working.

We need to set it up by making a new object. In addition, when we create the object, we have to tell it which input source we want. We will use the standard input (defaults to keyboard), which is specified by `System.in`. We can call this variable whatever we want. This syntax may look a little nicer if you completed the previous unit.

```
Scanner keyb = new Scanner(System.in);
```

This creates a new `Scanner` object, connects it to the keyboard, and stores a reference in the `keyb` variable.

If you try this, Java will say that it cannot find the command, unless you properly import the library. At the top of your program, `import java.util.*`, since `Scanner` is not a part of the basic command set.

Once Scanner is setup as above, you can then use the object to perform operations to read the keyboard (specifically, to read the standard system input).

Scanner has a whole collection of methods (commands tied to an object) built in. One such command is `nextLine()`, which pauses the program, and reads a line of input from standard in. It will pause until you hit enter after typing something in. ***IMPORTANT:*** you did not see this often in Processing, as the program often ran continuously and you just read the input state at any time. Here, the program pauses completely, and waits for user input. It un-pauses when the user hits enter to end their line of input.

This command returns a `String` that you can store somewhere or use in an operation.

```
String s = keyb.nextLine();
System.out.println("you typed: "+s);
```

You can keep re-using the object variable to get more input; don't create the object again, only create it once at the beginning of your program.

Here is my full program, including the library import, the class (with a name that matches the file name), the new main function, and all the new funny object-oriented syntax. Further, there is two cases of input on the same object variable.

```
import java.util.*;
public class FirstProgram {

      public static void main(String[] args)
      {
        Scanner keyb = new Scanner(System.in);
        System.out.println("type in the console: ");
        String s = keyb.nextLine();
        System.out.println("you typed: "+s);
        System.out.println("what is your name: ");
        String name = keyb.nextLine();
        System.out.println("your name is: "+name);
      }
}
```

Try to run this program, and once it's running, you can click on the console and type something in. After you press enter, the program will continue. In this case, it will echo back what you typed.

22.6 **Exercise: Echo loop**

Make a program that does the following:

- Asks the user for input, and gets a line from the user
- While the input is not empty (that is, not the empty string), echo's the output to

the console, and loop again.

This program should be pretty straight forward as you have now learned everything that you need to get it to work. The main gotcha is to realize that you need a while loop, since you don't know how many times the user will type in data. Try this on your own – setup a new program in Java from scratch – before looking at my solution:

```
import java.util.*;

public class FirstProgram
{
      static public void main(String[] args) {
            Scanner s = new Scanner(System.in);
            System.out.print("Give input:");
            String input = s.nextLine();
            while (!input.equals(""))
            {
                  System.out.println("You typed: "+input);
                  System.out.print("Give input:");
                  input = s.nextLine();
            }
        }
}
```

Can you guess how the `print` command works in comparison to the more familiar `println`?

One problem with this example is that we re-use some text, when really, they should be in global finals. In Java, you always put your code inside the class, so your final variables also need to be inside the class. There is on gotcha here – when you make a final constant, you need to add the `static` keyword on the declaration (see below). The reason for this is that we are not yet using the object oriented features. You will learn about that soon:

```
import java.util.*;

public class FirstProgram
{
      static final String PROMPT = "Give Input: ";

      static public void main(String[] args) {
            Scanner s = new Scanner(System.in);
            System.out.print(PROMPT);
            String input = s.nextLine();
            while (!input.equals(""))
            {
                  System.out.println("You typed: "+input);
                  System.out.print(PROMPT);
                  input = s.nextLine();
            }
      }
}
```

22.7 **Catch-up: some more commands**

There are some additional commands that you probably should know to get the most out of Java. First, you need to learn special commands to convert between numbers and strings. Your casts still work as expected, but Processing’s commands to do your string← → integer conversions do not work here. They have longer versions:

```
int Integer.parseInt(String)
```

For example:

```
import java.util.*;

public class FirstProgram
{
      static final String PROMPT = "Give Input: ";

      static public void main(String[] args) {
            Scanner s = new Scanner(System.in);
            System.out.print(PROMPT);
            String input = s.nextLine();
            int number = Integer.parseInt(input);
            System.out.println("The number is: "+number);
      }
}
```

Here, if you type in a number, it will convert it to an integer and then echo it out. Be careful, if you type something that is not a number, your program will crash. You will learn about error handling later.

Similarly, we have:

```
double Double.parseDouble(String)
```

and to convert in the other direction:

```
String Integer.toString(int)
String Double.toString(double)
```

`Scanner` also provides quite a few additional features that you should be able to use. First, take a look at the reference (see earlier in the chapter) and see if you can find any reasonable commands.

Some of the more common ones involve shortcuts of the above commands: instead of reading a String and then converting to a numerical format, you can read a number directly

```
Scanner s = new Scanner(System.in);
int i = s.nextInt();
double d = s.nextDouble();
```

Scanner gets a little more complex when you start using these commands. It will grab the next integer, but not necessarily up until the end of the line. For example, if you type in “15 20” and press enter, `nextInt` will give you 15.

Of particular power is scanner’s *tokenizing* functionality, the ability to break a string down into smaller chunks. For example, it can split a string on space, on comma, or whatever characters you want. It can look ahead before you convert to avoid crashes, and so on.

Another useful function of `Scanner` is that you can read from other sources. Instead of reading from `System.in` (the console keyboard), you can also read from files. Here is an example of reading the first line from a text file:

```
Scanner s = new Scanner(new File("myfile.txt"));
String input = s.nextLine();
System.out.println("The first line of the file: "+s);
```

To get this to work, you also need to import the `File` class from the `io` library, at the top of your program:

```
import java.io.File;
```

Scanner is quite powerful for processing text, but the full use is beyond the scope of this book. If you end up working with it, it is worth your time to read the reference pages on it. Two things to keep in mind are:

- Scanner's commands wait for a whole line of input, even if some commands only take part of the string
- Scanner will crash on unexpected input, until you learn error control.

22.8 Functions in Java

Applying what you have learned about functions to Java is very easy. First, in Java, they are called methods, not functions: methods are functions that are attached to an object. Since everything in Java is an object, then all the functions are methods.

There are only a few small things that you need to learn to get this to work. First, methods need to be placed inside the class, and not outside of it. It can go before or after the main method, it doesn't matter.

The second component is that you need to make your methods fit the object oriented model. Since you have not learned object oriented programming yet, you do not have the skills required to make the typical objects on methods. What you want to do is to make functions that work even without doing your object oriented programming. To do this, we need to add two keywords to the beginning of your function definition:

- `public`: anyone can access this
- `static`: the method always works, even when not in an *instantiated* object

Here is an example program. Here, we have a function that gets a line of input from the user. The function takes a scanner instance, and a prompt, and loops until the user enters some data:

```
import java.util.*;

public class FirstProgram
{
  static public String getInput(Scanner inputSource,
                                String prompt)
  {
      String s = "";
      do
      {
            System.out.print(prompt+" ");
            s = inputSource.nextLine();
      } while (s.equals(""));
      return s;
  }

  static public void main(String[] args)
  {
      Scanner kbd = new Scanner(System.in);
      String name = getInput(kbd, "What is your name?");
      int age = Integer.parseInt(getInput(kbd,
                                 "What is your age?"));
      String color = getInput(kbd,
                     "What is your favorite Color?");
      String output = "Hello "+name+" ("+age+" yrs), "+
                      "your favorite color is "+color+".";
      System.out.println(output);
  }
}
```

As usual, your functions can take and return any acceptable type in your program.

22.9 **Global variables in Java**

In processing, you used global variables extensively. This was required, as you had to remember and maintain state between calls to `draw`. In regular Java, there are no global variables in the same way.

You can emulate global variables by making non-final `static` variables inside the class and outside the functions, as we did for the finals earlier. However, you should not use this for now. Later, you will learn about instance variables and how to include them in your objects. For now, however, you really should get into the habit of passing everything that is required between your functions, and only using `final` constants in the global way. In fact, until you learn how to use them properly, you may lose marks for doing so.

The reality is that you do not need them for now. Since your `main` runs once and only once, and any looping needs to be managed by you, no state has to be remembered between calls. You can use your own local variables to do it. As you learn object oriented programming, you'll see better ways to solve your problems.

22.10 **Check your understanding: exercises**

You should be able to do all the exercises in this section without major effort. The actual programming logic itself should fall within what you learned in the course. Most of the new work falls under the new syntax of working with object-oriented programming. In addition, you have some basic new commands

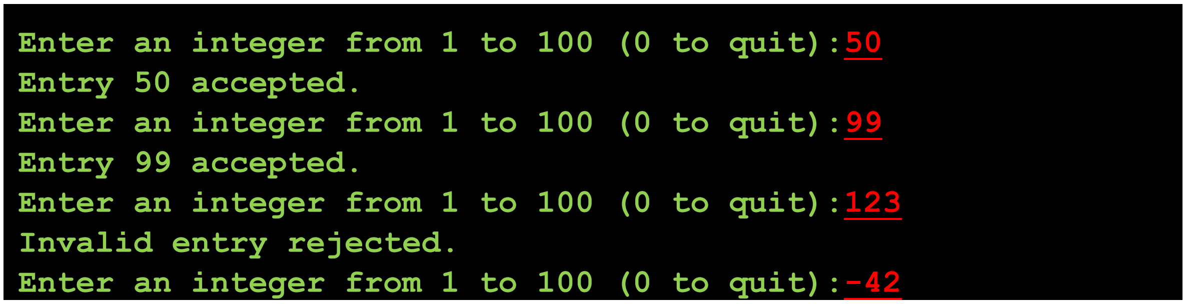
Exercise 1. Make a guessing game program, where the computer generates a random number and the user has to guess it..

a. Make `final` constants that determine the range that the random number can fall within, and how many guesses the user has.
b. In the main, call `Math.random` to generate a random number within the range.
c. In the main, Print out to the console some text telling them what the range is, and how many guesses they have.
d. Create a function that takes a prompt, an instantiated `Scanner` object, and returns an integer. This function asks the user to give a number, and returns their input as an integer. It does not do error checking.
e. In the main, use a `while` loop to…
   i. Ask the user to guess
   ii. Call your function to get the integer from the user
   iii. If the number is larger than the goal, say "too large", otherwise, say "too small"

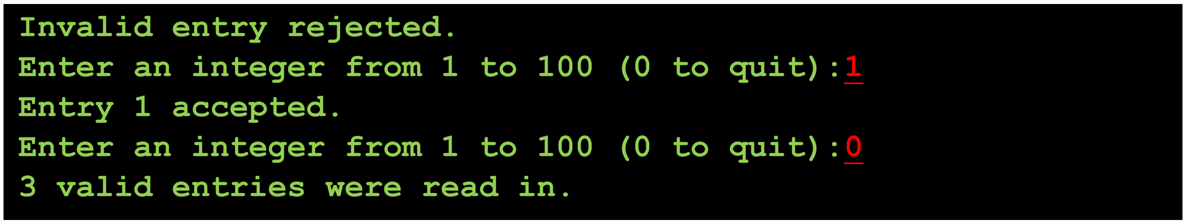
iv. Inform the user how many guesses they have left
v. Loop until the user is out of guesses, or they guess the number correctly.
f. Before the program ends, output the result, the number of used guesses, the actual number, etc.

Exercise 2. Make a program that works as follows. Create a method called `readData` which repeatedly prompts the user to enter an integer value. Values between 1 and 100 inclusive are "valid," and your method should print a message which echoes the value, and indicates that it is valid, as shown below. It should also keep track of the number of such valid values that were entered. The value 0 is special, and should cause your method to return without printing anything. Nothing should be printed out in this case. All other values should result in an error message being printed. When finished, return the number of valid values that were entered.

a. Make your `main` call this function, and your program should work as followed: note that user input is shown in red and underlined, and the final line was printed by `main`:



```
Enter an integer from 1 to 100 (0 to quit):50
Entry 50 accepted.
Enter an integer from 1 to 100 (0 to quit):99
Entry 99 accepted.
Enter an integer from 1 to 100 (0 to quit):123
Invalid entry rejected.
Enter an integer from 1 to 100 (0 to quit):-42
```



```
Invalid entry rejected.
Enter an integer from 1 to 100 (0 to quit):1
Entry 1 accepted.
Enter an integer from 1 to 100 (0 to quit):0
3 valid entries were read in.
```

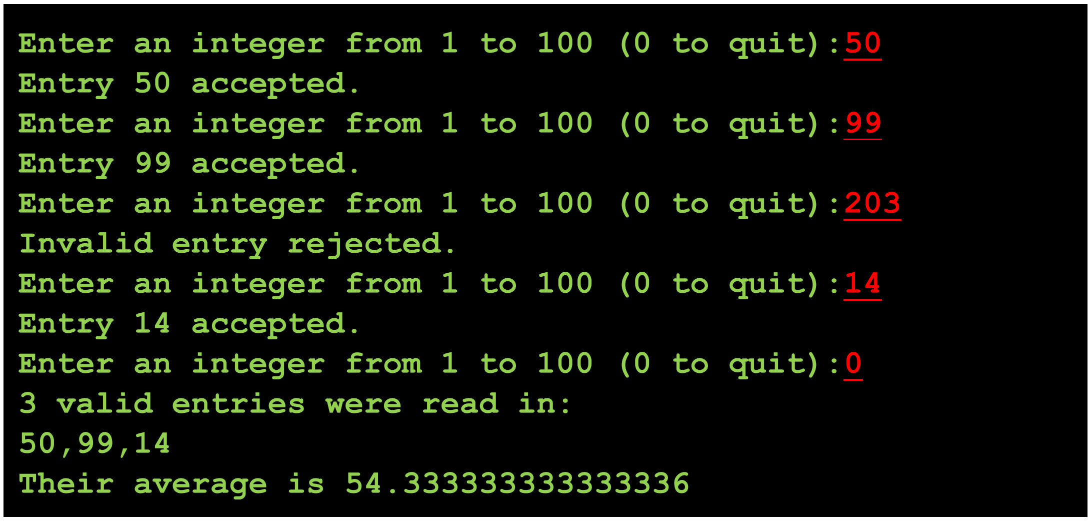
b. Update the program as follows: `readData` should accept a parameter of time `int[]`, where it should now store all the valid values. It should make sure that the array has enough room to store the next value, and if not, print an error message stating that there is not enough room.

   Create a method `printArray(int[] a, int n)` which prints the first `n` elements in the array, in the format shown below. The numbers should be separated by commas, but there should be no comma after the last one. There should be no blanks.

   Write a method `double average(int[] a, int n)` which will calculate and return the average of the first `n` items in `a`. Ve careful to return an accurate value calculated as a `double`, not as an `int`.

   Modify `main` to create an array that can hold up to 100 values. Use `readData` to populate the array, and then use `printArray` and `average` to generate your output.

   Here is an example:



```
Enter an integer from 1 to 100 (0 to quit):50
Entry 50 accepted.
Enter an integer from 1 to 100 (0 to quit):99
Entry 99 accepted.
Enter an integer from 1 to 100 (0 to quit):203
Invalid entry rejected.
Enter an integer from 1 to 100 (0 to quit):14
Entry 14 accepted.
Enter an integer from 1 to 100 (0 to quit):0
3 valid entries were read in:
50,99,14
Their average is 54.333333333333336
```

Exercise 3. Make a program that has the following methods:

`int[] getUserData()` – This method first asks the user how many integers they will provide. It then creates a new array with that many bins, and uses a loop to ask the user to provide integers for those bins. The method returns a reference to the array, that is now fully populated.

`int[] filterArray(int[] data, int min, int max)` – This method takes an array that is fully populated with data, and generates a new array that is filtered. The new array only contains elements from the first array that fall within `min` and `max` inclusive. One way to do this is to do two passes of the array: one to count the number of bins that will be needed. Then create the new array, and use a second pass to populate the new array. The new array is fully populated. Return a reference to the new array.

`String[] mapMagnitude(int[] data)` – This method takes an array of integers, and maps the values to strings representing the magnitude of the value. These strings should be stored into a new array with the same length as `data`, with the values corresponding. For example, bin `i` in the new array should correspond to bin `i` in the `data` array. Values from 0...9 should give “one”, 10...99 should give “ten”, 100…999 should give “one hundred”, …, 1,000,000…9,999,999 should give “million”, and so on. Negative values all should give “negative”. Use your knowledge of the possible range of the integer type to determine how large to go.

In addition, create `printIntArray` and `printStringArray` to help you see the results.

Your `main` should call `getUserData` to get an input array, and then call both the filtering and mapping methods with the data, and your printing methods to test your result.

Exercise 4. As you learned in the course, binary search requires that data is sorted. However, when you get data from a user, how can you guarantee that it is sorted? You will make a method called `insertSort` that adds an element to an array, guaranteeing that the resulting array is sorted.

This will use partially-filled arrays, as your method will need to know how many bins of the array so far are filled with data (and assumed to be sorted). As such, your method header will be as follows:

`int insertSort(int[] data, int size, int newElem)`. This method assumes that `data` is already sorted, and has `size` elements. It will insert `newElem` into the array, such that the size is one larger, and the array is now sorted. The new partially-filled size of the array is returned. The algorithm works as follows:

- When `size` is zero, the new element is placed in the first bin as the only element
- Otherwise, use a loop to go through the first `size` elements of the array, with a variable `toInsert` initially set to `newElem`, the data to insert.
  - If the data at bin `i` is less than `toInsert`, do nothing, since `toInsert` belongs to the right of this position.
  - If the data at bin `i` is larger than `toInsert`, then swap `toInsert` with the value currently at the bin. That is, insert our new value here, and then take the existing data out, and see where it belongs next.
- The above two checks are sufficient. Once the correct location is found, then the remaining elements bubble to the right.

Make a program that uses a `while` loop to prompt the user for numbers, and your new method to place them in an array. Print out the array as you go to ensure that it is always sorted as you add data.

A great source for basic Java practice is the University of Manitoba's list of historical programming contest questions for the high-school level. These, particularly the easier ones, should be manageable (with some work) by students at this level.

http://cs.umanitoba.ca/highschool/past-contests.php

http://www.cs.umanitoba.ca/~acmpc/past/

(this page intentionally left blank)

UNIT 23. USING PROCESSING IN JAVA

For the most part, you will end up leaving much of Processing behind as you move to pure Java. However, Processing is very useful in how much power it gives you over graphics and user input. As such, you may still want to use your Processing knowledge to add graphics to your programs.

To get Processing to work inside Java, you have a few problems to solve. The main problem is to import all that Processing capability into your Java system. Luckily, you can do this by adding a library. Then, you need to learn how to write that code into your program.

Unfortunately, how to link an external library to you project is highly specific to the actual IDE that you use. If you dig around, you should be able to find out how to link a JAR file – a Java ARchive.

In Eclipse, it is quite simple. Start a new project as per the code above. In the Package Explorer in the left-hand-side of Eclipse, right click on your project name (the top-level tree hierarchy); for example, `FirstProject`. Look for the item in the menu called Build Path – this tells Java which files to include in the project. Don't use the `import` command as this actually copies everything fully into your project, and simply won't work.

Click on Build Path and then click Add External Archives. A window should pop up that lets you select a file.

Navigate to your Processing folder where you installed it on your computer. Once there, go into the `core` subfolder and then `library`. Select the `core.jar` file and open, to add it to your project. You have just include the core Processing capability into your Java project!

You can test this by trying to `import` a Processing library into your project. Open the `FirstProgram` source file, and at the very top, add the following line:

```
import processing.core.PApplet;
```

Try to run your program. If you get an error (e.g., that the import cannot be resolved), then your Processing JAR file is not properly linked. Try again or ask for help.

If your program works without complaint, that means that your import worked, and you now have access to Processing functionality!

To get Processing to work, we need to do a whole bunch of things. As this is unfortunately complex and requires things you haven't learned yet, my advice is to copy-paste my boiler-plate code below. However, at least try to read through the following changes:

- integrate your code into the Processing object model. Unfortunately this uses new object oriented programming techniques that you haven't learned. You need to make your class *extend* the existing Process class;
- modify `main` to get it to start the Processing engine, which unfortunately uses an advanced technique called *reflection* that you haven't learned yet;
- put your regular Processing functions inside this new class, and also put the word `public` in front of them, for object-oriented reasons you haven't learned yet;
- Put your `size` command (to set the canvas size) not in the `setup`, but instead in a new function called `settings`.

If you do all this, your program should work. Since you are extending the `PApplet`, your familiar Processing commands are freely available within the class.

Here is the boiler plate:

```
import processing.core.PApplet;

public class FirstProgram extends PApplet
{
      public void settings()
      {
            size(500,500);
      }

      public void setup()
      {
      }

      public void draw()
      {
      }

      static public void main(String[] args) {
         String[] appletArgs =
                new String[] {"FirstProgram"};
         PApplet.main(appletArgs);
        }
}
```

The only gotcha here is that the string used in the main MUST match the name of your class. That's a little annoying, but can't be helped.

Also, there is likely some confusion as to how to integrate other concepts you have learned, so here is a broader example including arrays, user defined functions, etc.

```
import processing.core.PApplet;

public class FirstProgram extends PApplet
{
      final int POINTS = 100;
      final int BEAM_SIZE = 50;
      final int SIZE = 2;
      float[] x = new float[POINTS];
      float[] y = new float[POINTS];
      public void settings()
      {
            size(500,500);
      }

      public void setup()
      {
            for (int i = 0; i < POINTS; i++)
            {
                  x[i] = random(width);
                  y[i] = random(BEAM_SIZE)-BEAM_SIZE/2;
            }
      }

      public void drawBeam()
      {
            for (int i = 1; i < POINTS; i++)
            {

              ellipse(x[i]+mouseX, y[i]+mouseY,SIZE,SIZE);
              x[i] = (x[i]*x[i]+y[i])%width;
            }
      }

      public void draw()
      {
            background(0);
            stroke(255);
            drawBeam();
      }

      static public void main(String[] args) {
         String[] appletArgs =
                    new String[] {"FirstProgram"};
         PApplet.main(appletArgs);
        }
}
```