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.

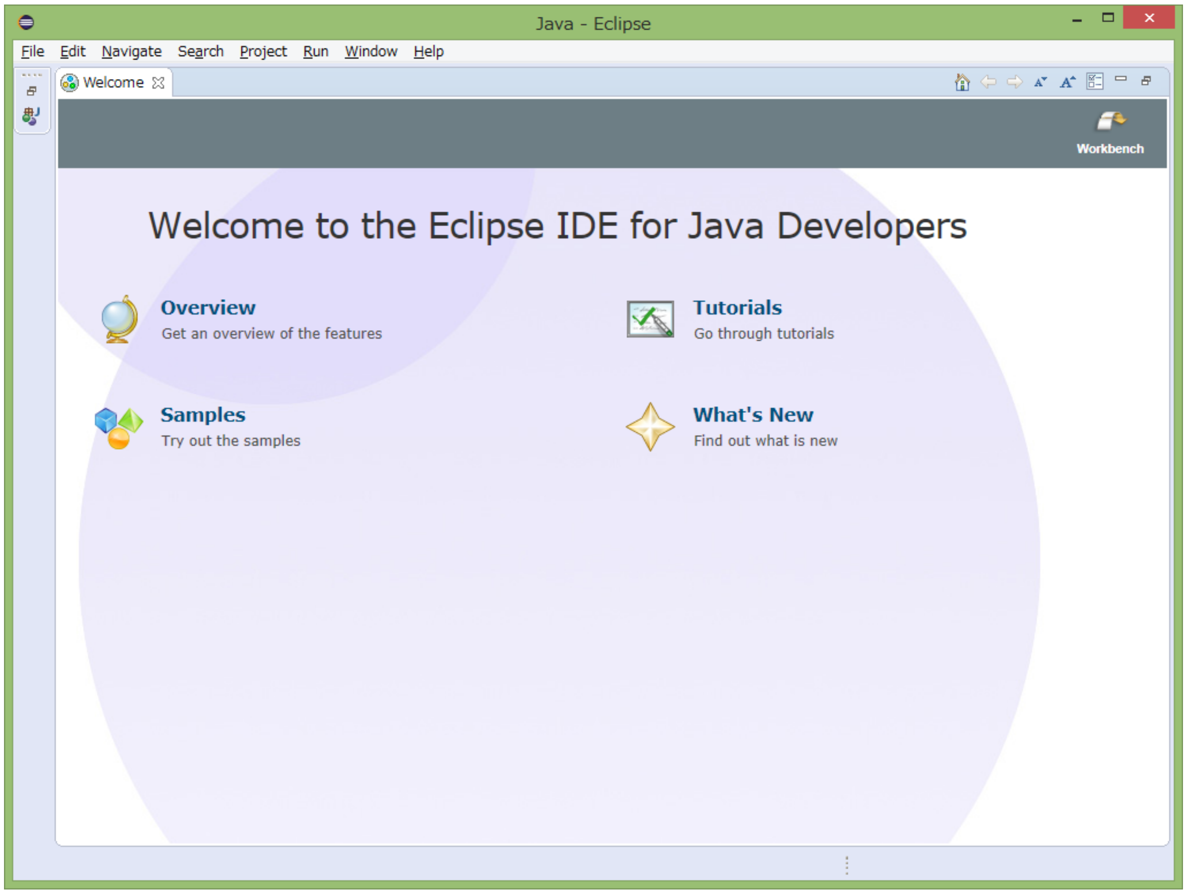
- **Dr Java** is a teaching tool, a Java editor for learners and beginners.
- **Textpad** is a similar tool, but for windows only.
- **Eclipse** is a professional tool for Java development.
- **BlueJ** is the official editor of Green Foot, which is another graphics system for Java
- **Sublime** is a 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

To get started, click the “Workbench” icon on the top right.

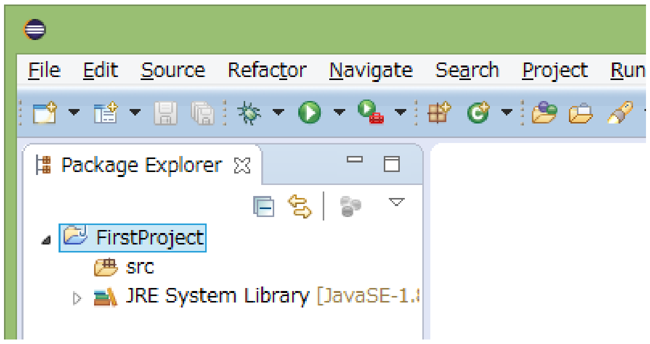
Then, you want to create a new project. It is a good idea to save each assignment, lab, etc., as its own project. Inside one project, files can conflict with each other.

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).

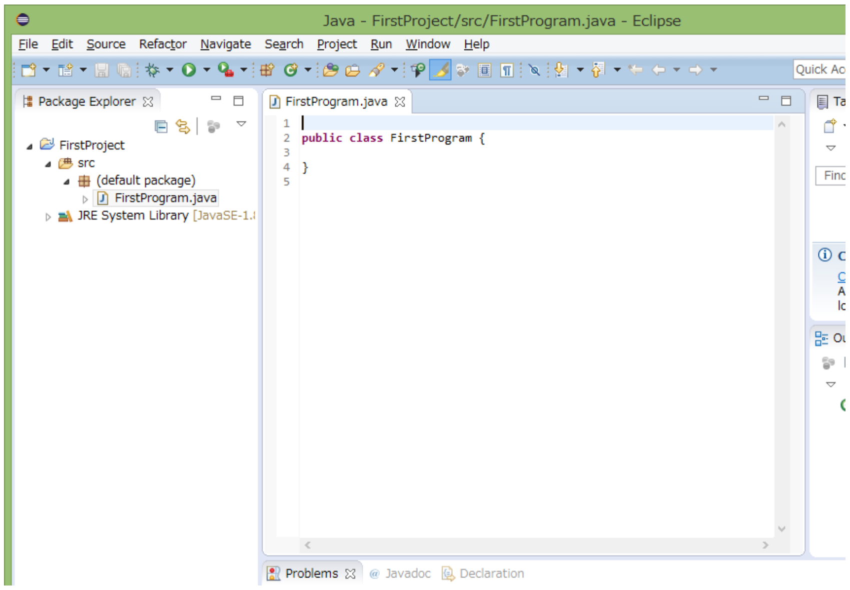
A pop up window comes up with all kinds of settings. All that you need to do is to create a project name.

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)