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)
-

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.