Chapter 1

Programming in Java

What is in This Chapter ?

This first chapter introduces you to programming JAVA applications. It assumes that you are already familiar with programming and that you have taken either Processing or Python in the previous course. You will learn here the basics of the JAVA language syntax. In this chapter, we discuss how to make JAVA applications and the various differences between JAVA applications and the Processing/Python applications that you may be used to writing.

1.1 Object-Oriented Programming and JAVA

Object-oriented programming (OOP) is a way of programming in which your code is organized into objects that interact with one another to form an application. When doing OOP, the programmer (i.e., you) spends much time ***defining*** (i.e., writing code for) various objects by specifying the attributes (or data) that make up the object as well as small/simple functional behaviors that the object will need to respond to (e.g., deposit, withdraw, compute interest, get age, save data etc...)

There is nothing *magical* about OOP. Programmers have been coding for years in traditional top/down structured programming languages. So what is so great about OO-Programming ? Well, OOP uses 3 main powerful concepts:

Inheritance

- promotes code sharing and re-usability
- intuitive hierarchical code organization

Encapsulation

- provides notion of security for objects
- reduces maintenance headaches
- more robust code

Polymorphism

- simplifies code understanding
- standardizes method naming

We will discuss these concepts later in the course once we are familiar with JAVA .

Through these powerful concepts, object-oriented code is typically:

- **easier to understand** (relates to real world objects)
- better **organized** and hence easier to work with
- **simpler** and **smaller** in size
- more **modular** (made up of plug-n'-play re-usable pieces)
- better **quality**

This leads to:

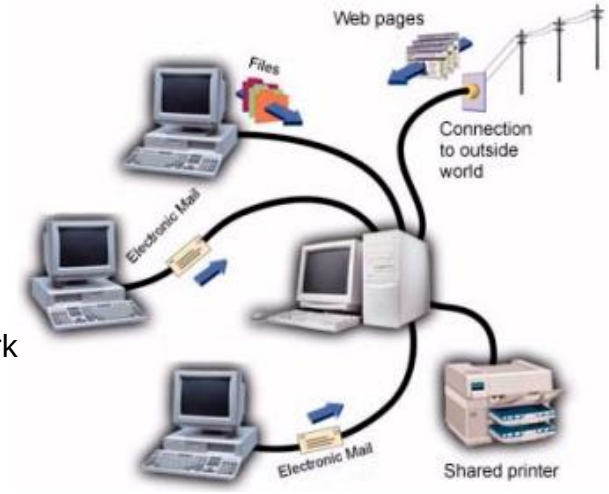
- high productivity and a **shorter delivery cycle**
- **less manpower** required
- **reduced costs** for maintenance
- more **reliable** and **robust** software
- **pluggable** systems (updated UI's, less legacy code)

JAVA is a very popular object-oriented programming language from SUN Microsystems. In the previous course you may have used a language called **Processing** which is written "on top of" JAVA. That is, **Processing** uses the same syntax as JAVA. However **Processing** simplifies the process required to get a program "up and running" in that some of the overhead code is hidden from the programmer. In addition, some of the functionality in **Processing** has been simplified such as graphics and event handling ... which are just a little more complicated in JAVA.

JAVA has become a basis for new technologies such as: Enterprise Java Beans (EJB's), Servlets and Java Server Pages (JSPs) , etc. In addition, many packages have been added which extend the language to provide special features:

- **Java Media Framework** (for video streaming, webcams, MP3 files, etc)
- **Java 3D** (for 3D graphics)
- **Java Advanced Imaging** (for image manipulation)
- **Java Speech** (for dictation systems and speech synthesis)
- **Java FX** (for graphics, web apps, charts/forms, etc..)
- **J2ME** (for mobile devices such as cell phones)
- **Java Embedded** (for embedding java into hardware to create smart devices)

JAVA is continually changing/growing. Each new release fixes bugs and adds features. New technologies are continually being incorporated into JAVA. Many new packages are available. Just take a look at the **www.oracle.com/technetwork/java/index.html** website for the latest updates. There are many reasons to use JAVA:



- **architecture independence**
  - ideal for internet applications
  - code written once, runs anywhere
  - reduces cost $$$
- **distributed and multi-threaded**
  - useful for internet applications
  - programs can communicate over network
- **dynamic**
  - code loaded only when needed
- **memory managed**
  - automatic memory allocation / de-allocation
  - garbage collector releases memory for unused objects
  - simpler code & less debugging

- **robust**
  - strongly typed
  - automatic bounds checking
  - no "pointers" (you will understand this in when you do **C** language programming)

The JAVA programming language itself (i.e., the SDK (Software Development Kit) that you download from SUN) actually consists of many program pieces (or object class definitions) which are organized in groups called ***packages*** (i.e., similar to the concept of ***libraries*** in other languages) which we can use in our own programs.

When programming in JAVA, you will usually use:

- classes from the JAVA class libraries (used as *tools*)
- classes that you will create yourself
- classes that other people make available to you

Using the JAVA class libraries whenever possible is a good idea since:

- the classes are carefully written and are efficient.
- it would be silly to write code that is already available to you.

We can actually create our own packages as well, but this will not be discussed in this course.

How do you get started in JAVA?

When you download and install the latest **JAVA SDK**, you will not see any particular application that you can run which will bring up a window that you can start to make programs in. That is because the SUN guys, only supply the JAVA SDK which is simply the compiler and virtual machine. JAVA programs are just text files, they can be written in any type of text editor. Using a most rudimentary approach, you can actually open up windows **NotePad** and write your program ... then compile it using the windows **Command Prompt** window. This can be tedious and annoying since JAVA programs usually require you to write and compile multiple files.

A better approach is to use an additional piece of application software called an ***Integrated Development Environment (IDE)***. Such applications allow you to:

- write your code with colored/formatted text
- compile and run your code
- browse java documentation
- create user interfaces visually
- and use other java technologies (e.g. Java Beans, EJB's, Servlet programming etc...)

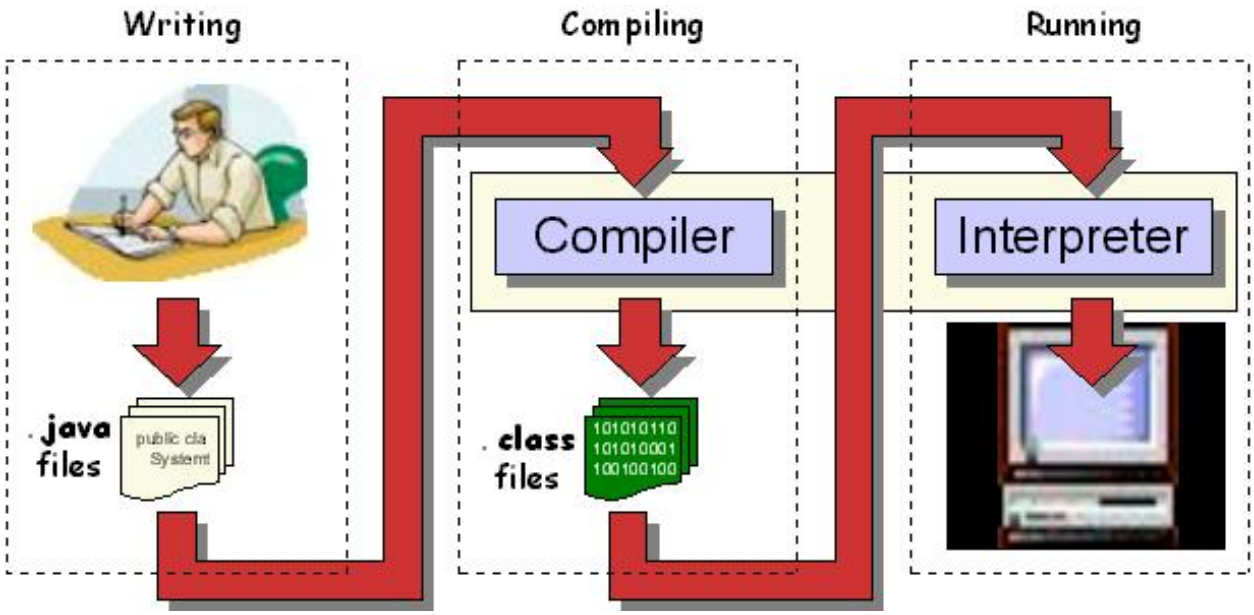
There are many IDE's that you can use. You may choose whatever you wish. Here are a few:

- **JCreator LE** (Windows) - download from **www.jcreator.com**
- **JGrasp** (Windows, Mac OS X, Linux) - download from **www.jgrasp.com**
- **Eclipse** (Windows, Mac OS X, Linux) - download from **www.eclipse.org**
- **Dr. Java** (Windows, Mac OS X) - download from **drjava.sourceforge.net**

1.2 Writing Your First JAVA Program

The process of writing and using a JAVA program is as follows:

1. **Writing:** define your classes by writing what is called **.java** files (a.k.a. ***source code***).
2. **Compiling:** send these **.java** files to the JAVA compiler, which will produce **.class** files
3. **Running:** send one of these **.class** files to the JAVA interpreter to run your program.



The java ***compiler***:

- prepares your program for running
- produces a **.class** file containing ***byte-codes*** (which is a program that is ready to run).

If there were errors during compiling (i.e., called "***compile-time***" errors), you must then fix these problems in your program and then try compiling it again.

The java ***interpreter*** (a.k.a. ***Java Virtual Machine (JVM)***):

- is required to run any JAVA program
- reads in **.class** files (containing byte codes) and translates them into a language that the computer can understand, possibly storing data values as the program executes.

Just before running a program, JAVA uses a ***class loader*** to put the byte codes in the computer's memory for all the classes that will be used by the program. If the program produces errors when run (i.e., called "***run-time***" errors), then you must make changes to the program and re-compile again.

Our First Program

The first step in using any new programming language is to understand how to write a simple program. By convention, the most common program to begin with is always the "hello world" program which when run ... should output the words "Hello World" to the computer screen. We will describe how to do this now. When compared to Processing, you will notice that JAVA requires a little bit of overhead (i.e., extra code) in order to get a program to run.

All of your programs will consist of one or more files called ***classes***. Last term we defined classes only to represent a data structure with some variables in it. However, in JAVA, each time you want to make any program, you need to define a class. That means, each program requires us to define a data structure (or object), although sometimes we will not even define any data (or variables) for the object.

Here is the first program that we will write:

```
public class HelloWorldProgram {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Here are a few points of interest in regards to ALL of the programs that you will write in this course:

- The program must be saved in a file with the same name as the class name (spelled the same exactly with upper/lower case letters and with a **.java** file extension). In this case, the file must be called **HelloWorldProgram.java**.
- The first line beings with words **public class** and then is followed by the name of the program (which must match the file name, except not including the .java extension). The word **public** indicates that this will be a "publically visible" class definition that we can run from anywhere. We will discuss this more later.
- The entire class is defined within the first opening brace **{** at the end of the first line and the last closing brace **}** on the last line.
- The 2nd line (i.e., **public static void main(String args[]) {**) defines the starting place for your program and will ALWAYS look exactly as shown. In Processing, the starting place for our program was simply the top line of the program and then **setup()** was called, followed by an infinite loop that called the **draw()** procedure. There are NO **setup()** or **draw()** procedures in JAVA. Instead, the program always starts running by calling this **main()** procedure which takes a String array as an incoming parameter. This String array represents what are called "command-line-arguments" which allows you to start the program with various parameters. However, we will not use these parameters in the course and so we will not discuss it further.
- The 2nd last line will be a closing brace **}**.

So … ignoring the necessary "template" lines, the actual program consists of only one line: **System.out.println("Hello World");** which actually prints out the characters **Hello World** to the screen. You may recall that this was a little simpler in Processing since we simply did **println("Hello World");**

So ... to summarize, every java program that you will write will have the following basic format:

```
public class ____________________ {
    public static void main(String[] args) {
        ____________________________;
        ____________________________;
        ____________________________;
    }
}
```

Just remember that YOU get to pick the program name (e.g., **MyProgram**) which should ALWAYS start with a capital letter. Also, your code MUST be stored in a file with the same name (e.g., **MyProgram.java**). Then, you can add as many lines of code as you would like in between the inner **{ }** braces. You should ALWAYS line up ALL of your brackets using the **Tab** key on the keyboard.

In Processing, all applications ran with a graphical window that allowed us to display graphics and text as well as get user input via the keyboard and mouse. In JAVA however, there is no such window that pops up. Instead, any program output simply appears in a System console, which is usually a pane in the IDE's window.

Later in the course, we will create our own windows. For now, however, we will simply use the System console to display results. This will allow us to focus on understanding what is going on "behind the scenes" of a windowed application. It is important that we first understand the principles of Object-Oriented Programming.

1.3 Python vs. Processing vs. Java

Although **Processing** is based on **JAVA** syntax, it uses simplified names for functions and procedures. **Python** code differs even more in regards to syntax. Provided here is a brief explanation of a few of the differences (and similarities) between the three languages:

Commenting Code:

Python	**Processing & JAVA** (i.e., they are the same)
`# single line comment`	`// single line comment`

`""" a multiline comment`
`which spans more`
`than one line.`
`"""`
	`/* a multiline comment`
	`which spans more`
	`than one line.`
	`*/`

Displaying Information To the System Console:

Python	`print 'The avg is ', avg` `print 'All Done'`
Processing	`println ("The avg is " + avg);` `println ("All Done");`
Java	`System.out.println("The avg is " + avg);` `System.out.println("All Done");`

Math Functions:

Python	Processing	Java

`min(a, b)`
`max(a, b)`
`round(a)`
`pow(a, b)`
`sqrt(a)`
`abs(a)`
`sin(a)`
`cos(a)`
`tan(a)`
`degrees(r)`
`radians(d)`
`random.random()`
	`min(a, b)`
	`max(a, b)`
	`round(a)`
	`pow(a, b)`
	`sqrt(a)`
	`abs(a)`
	`sin(a)`
	`cos(a)`
	`tan(a)`
	`degrees(r)`
	`radians(d)`
	`random(n)`
		`Math.min(a, b)`
		`Math.max(a, b)`
		`Math.round(a)`
		`Math.pow(a, b)`
		`Math.sqrt(a)`
		`Math.abs(a)`
		`Math.sin(a)`
		`Math.cos(a)`
		`Math.tan(a)`
		`Math.toDegrees(r)`
		`Math.toRadians(d)`
		`Math.random()`

Variables:

Python	Processing	Java

`hungry = True;`
`days = 15;`
`age = 19;`
`years = 3467;`
`seconds = 1710239;`
`gender = 'M';`
`amount = 21.3;`
`weight = 165.23;`
	`boolean hungry = true;`
	`int days = 15;`
	`byte age = 19;`
	`short years = 3467;`
	`long seconds = 1710239;`
	`char gender = 'M';`
	`float amount = 21.3;`
	`double weight = 165.23;`
		`boolean hungry = true;`
		`int days = 15;`
		`byte age = 19;`
		`short years = 3467;`
		`long seconds = 1710239;`
		`char gender = 'M';`
		`float amount = 21.3f;`
		`double weight = 165.23;`

Constants:

Python	Processing	Java

`do not make our own by default`
 	
`math.pi`
	`final int DAYS = 365;`
	`final float RATE = 4.923;`
	
	`PI`
		`final int DAYS = 365;`
		`final float RATE = 4.923f;`
		
		`Math.PI`

Type Conversion:

Python	Processing	Java

`d = 65.237898546;`
`f = float(d);`
`i = int(f);`
`g = long(i);`
`c = chr(i);`
	`double d = 65.237898546;`
	`float f = (float)d;`
	`int i = int(f);`
	`float g = float(i);`
	`char c = char(i);`
		`double d = 65.237898546;`
		`float f = (float)d;`
		`int i = (int)f;`
		`float g = (float)i;`
		`char c = (char)i;`

Arrays:

Python	**Processing & Java** (i.e., they are the same)

`days = zeros(30, Int)`
`weights = zeros(100, Float)`
`names = [];`
`rentals = [];`
`friends = [];`
 	
`ages = [34, 12, 45]`
`weights = [4.5,2.6,1.5]`
`names = ['Bill', 'Jen']`
	`int[] days = new int[30];`
	`double[] weights = new double[100];`
	`String[] names = new String[3];`
	`Car[] rentals = new Car[500];`
	`Person[] friends = new Person[50];`
	
	`int[] ages = {34, 12, 45};`
	`double[] weights = {4.5,2.6,1.5};`
	`String[] names = {"Bill", "Jen"};`

FOR loops:

Python	Processing	Java

`total = 0`
`for i in range (1, n):`
`  total += i`
 	
`print total`
	`int total = 0;`
	`for (int i=1; i<=n; i++) {`
	`  total += i;`
	`}`
	`println(total);`
		`int total = 0;`
		`for (int i=1; i<=n; i++) {`
		`  total += i;`
		`}`
		`System.out.println(total);`

WHILE loops:

Python	Processing	Java

`speed = 0`
`x = 0`
`while x <= width:`
`  speed = speed + 2`
`  x = x + speed`
	`int speed = 0;`
	`int x=0;`
	`while (x <= width) {`
	`  speed = speed + 2;`
	`  x = x + speed;`
	`}`
		`int speed = 0;`
		`int x=0;`
		`while (x <= width) {`
		`  speed = speed + 2;`
		`  x = x + speed;`
		`}`

IF statements:

Python

```
if (grade >= 80) and (grade <=100):
    print 'Super!'

if grade >= 50:
    print grade
    print 'Passed!'
else:
    print 'Grade too low.'
```

Processing

```
if ((grade >= 80) && (grade <=100))
    println("Super!");

if (grade >= 50) {
    println(grade);
    println("Passed!");
}
else
    println("Grade too low.");
```

Java

```
if ((grade >= 80) && (grade <=100))
    System.out.println("Super!");

if (grade >= 50) {
    System.out.println(grade);
    System.out.println("Passed!");
}
else
   System.out.println("Grade too low.");
```

Procedures & Functions:

Processing	**Processing** and **Java**

`def procName(x, c):`
`  // Write code here`
 	
`def funcName(h):`
`  result = ....`
`  // Write code here`
`  return result`
	`void procName(int x, char c)  {`
	`  // Write code here`
	`}`
	
	`double funcName(float h) {`
	`  result = ....;`
	`  // Write code here`
	`  return result;`
	`}`

As the course continues, you will notice other differences between **Python**, **Processing** and **JAVA**. However, the underlying programming concepts remain the same. As we do coding examples throughout the course, you will get to know the some of the other intricate details of basic JAVA syntax. Therefore, we will not discuss this any further at this point.

1.4 Getting User Input

In **Processing**, all applications ran with a graphical window that allowed us to display graphics and text as well as get user input via the keyboard and mouse. In JAVA however, there is no such window that pops up automatically.

In addition to outputting information to the console window, JAVA has the capability to get input from the user. Unfortunately, things are a little "messier/uglier" when getting input. The class is called **Scanner** and it is available in the **java.util** package (more on packages later).

To get input from the user, we will create a new **Scanner** *object* for input from the **System** console. Here is the line of code that gets a line of text from the user:

```
new Scanner(System.in).nextLine();
```

This line of code will wait for the user (i.e., you) to enter some text characters using the keyboard. It actually waits until you press the **Enter** key. Then, it returns to you the characters that you typed (not including the **Enter** key). You can then do something with the characters, such as print them out.

Here is a simple program that asks users for their name and then says hello to them:

```
import java.util.Scanner;    // More on this later

public class GreetingProgram {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("What is your name ?");
        System.out.println("Hello, " + keyboard.nextLine());
    }
}
```

Notice the output from this program if the letters Mark are entered by the user (Note that the blue text (i.e., 2nd line) was entered by the user and was not printed out by the program):

```
What is your name ?
Mark
Hello, Mark
```

As you can see, the **Scanner** portion of the code gets the input from the user and then combines the entered characters by preceding it with the **"Hello, "** string before printing to the console on the second line.

Interestingly, we can also read in integers from the keyboard as well by using the **`nextInt()`** function instead of **`nextLine()`**. For example, consider this calculator program that finds the average of three numbers entered by the user:

```
import java.util.Scanner;    // More on this later

public class CalculatorProgram {
    public static void main(String[] args) {
        int sum;

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter three numbers:");
        sum = keyboard.nextInt() + keyboard.nextInt() + keyboard.nextInt();
        System.out.println("The average of these numbers is " + (sum/3.0));
    }
}
```

Here is the output when the **CalculatorProgram** is run with the numbers 34, 89 and 17 entered:

```
Enter three numbers:
34
89
17
The average of these numbers is 46.666666666666664
```

There is much more we can learn about the **Scanner** class. It allows for quite a bit of flexibility in reading input. In place of **nextLine()**, we could have used any one of the following to specify the kind of primitive data value that we would like to get from the user:

```
nextInt(), nextShort(), nextLong(), nextByte(), nextFloat(),
nextDouble(), nextBoolean(), next()
```

Notice that there is no **nextChar()** function available. The **next()** function actually returns a String of characters, just like **nextLine()**. If you wanted to read a single character from the keyboard (but don't forget that we still need to also press the **Enter** key), you could use the following: **next().charAt(0)**. We will look more into this later when we discuss **String** functions. It is important to use the correct function to get user input. For example, if we were to enter 10, 20 into our program above, followed by some "junk" characters ... an error will occur telling us that there was a problem with the input as follows:

```
java.util.InputMismatchException
      ...
      at java.util.Scanner.nextInt(Unknown Source)
      at BetterCalculatorProgram.main(BetterCalculatorProgram.java:11)
      ...
```

This is JAVA's way of telling us that something bad just happened. It is called an ***Exception***. We will discuss more about this later. For now, assume that valid integers are entered.

Example:

Let us write a program that displays the following menu.

```
Luigi's Pizza
-------------------------------------------
                     S(SML)  M(MED)  L(LRG)
1. Cheese             5.00    7.50   10.00
2. Pepperoni          5.75    8.63   11.50
3. Combination        6.50    9.75   13.00
4. Vegetarian         7.25   10.88   14.50
5. Meat Lovers        8.00   12.00   16.00
```

The program should then prompt the user for the type of pizza he/she wants to order (i.e., 1 to 5) and then the size of pizza 'S', 'M' or 'L'. Then the program should display the cost of the pizza with 13% tax added.

To begin, we need to define a class to represent the program and display the menu:

```
public class LuigisPizzaProgram {
      public static void main(String args[]) {
            System.out.println("Luigi's Pizza                                  ");
            System.out.println("-----------------------------------------------");
            System.out.println("                      S(SML)  M(MED)  L(LRG)");
            System.out.println("1. Cheese              5.00    7.50   10.00 ");
            System.out.println("2. Pepperoni           5.75    8.63   11.50 ");
            System.out.println("3. Combination         6.50    9.75   13.00 ");
            System.out.println("4. Vegetarian          7.25   10.88   14.50 ");
            System.out.println("5. Meat Lovers         8.00   12.00   16.00 ");
      }
}
```

We can then get the user input and store it into variables. We just need to add these lines (making sure to put **`import`** `java.util.Scanner;` at the top of the program):

```
Scanner keyboard = new Scanner(System.in);

System.out.println("What kind of pizza do you want (1-5) ?");
int kind = keyboard.nextInt();

System.out.println("What size of pizza do you want (S, M, L) ?");
char size = keyboard.next().charAt(0);
```

Now that we have the **kind** and **size**, we can compute the total cost. Notice that the cost of a small pizza increases by $0.75 as the kind of pizza increases. Also, you may notice that the cost of a medium is 1.5 x the cost of a small and the cost of a large is 2 x a small. So we can compute the cost of any pizza based on its kind and size by using a single mathematical formula. Can you figure out the formula ?

A small pizza would cost: **smallCost** = $4.25 + (**kind** x $0.75)
A medium pizza would cost: **mediumCost** =**smallCost** * 1.5
A large pizza would cost: **largeCost** =**smallCost** * 2.

Can you write the code now ?

```
float cost = 4.25f + (kind * 0.75f);
if (size == 'M')
      cost *= 1.5f;
else if (size == 'L')
      cost *= 2;
```

And of course, we can then compute and display the cost before and after taxes. Here is the completed program:

```
import java.util.Scanner;

public class LuigisPizzaProgram {
	public static void main(String args[]) {
		System.out.println("Luigi's Pizza                                    ");
		System.out.println("-----------------------------------------------");
		System.out.println("                      S(SML)  M(MED)  L(LRG)");
		System.out.println("1. Cheese              5.00    7.50   10.00 ");
		System.out.println("2. Pepperoni           5.75    8.63   11.50 ");
		System.out.println("3. Combination         6.50    9.75   13.00 ");
		System.out.println("4. Vegetarian          7.25   10.88   14.50 ");
		System.out.println("5. Meat Lovers         8.00   12.00   16.00 ");

		Scanner keyboard = new Scanner(System.in);

		System.out.println("What kind of pizza do you want (1-5) ?");
		int kind = keyboard.nextInt();

		System.out.println("What size of pizza do you want (S, M, L) ?");
		char size = keyboard.next().charAt(0);

		float cost = 4.25f + (kind * 0.75f);
		if (size == 'M')
			cost *= 1.5f;
		else if (size == 'L')
			cost *= 2;

		System.out.println("The cost of the pizza is: $" + cost);
		System.out.println("The price with tax is: $" + cost*1.13);
	}
}
```

The above program displays the price of the pizza quite poorly. For example, here is the output of we wanted a Large Cheese pizza:

```
The cost of the pizza is: $5.0
The price with tax is: $5.6499999999999995
```

It would be nice to display money values with proper formatting (i.e., always with 2 decimal places). The next section will cover this.

1.5 Formatting Text

Consider the following similar program which asks the user for the **price** of a product, then displays the **cost** with **taxes** included, then asks for the **payment** amount and finally prints out the **change** that would be returned:

```
import java.util.Scanner;

public class ChangeCalculatorProgram {
    public static void main(String[] args) {
        // Declare the variables that we will be using
        double  price, total, payment, change;

        // Get the price from the user
        System.out.println("Enter product price:");
        price = new Scanner(System.in).nextFloat();

        // Compute and display the total with 13% tax
        total = price * 1.13;
        System.out.println("Total cost:$" + total);

        // Ask for the payment amount
        System.out.println("Enter payment amount:");
        payment = new Scanner(System.in).nextFloat();

        // Compute and display the resulting change
        change = payment - total;
        System.out.println("Change:$" + change);
    }
}
```

Here is the output from running this program with a price of $**35.99** and payment of $**50**:

```
Enter product price:
35.99
Total cost:$40.66870172505378
Enter payment amount:
50
Change:$9.33129827494622
```

Notice all of the decimal places. This is not pretty. Even worse …if you were to run the program and enter a price of **8.85** and payment of **10**, the output would be as follows:

```
Enter product price:
8.85
Total cost:$10.0005003888607
Enter payment amount:
10
Change:$-5.003888607006957E-4
```

The **E-4** indicates that the decimal place should be moved 4 units to the left…so the resulting change is actually -$0.0005003888607006957. While the above answers are correct, it would be nice to display the numbers properly as numbers with 2 decimal places.

JAVA's **String** class has a nice function called **format()** which will allow us to format a String in almost any way that we want to. Consider (from our code above) replacing the change output line to:

```
System.out.println("Change:$" + String.format("%,1.2f", change));
```

The **String.format()** always returns a **String** object with a format that we get to specify. In our example, this **String** will represent the formatted **change** which is then printed out. Notice that the function allows us to *pass-in* two parameters (i.e., two pieces of information separated by a comma **,** character). Recall that we discussed parameters when we created constructors and methods for our own objects.

The first parameter is itself a **String** object that specifies how we want to format the resulting String. The second parameter is the value that we want to format (usually a variable name). Pay careful attention to the brackets. Clearly, **change** is the variable we want to format. Notice the format string **"%,1.2f"**. These characters have special meaning to JAVA. The **%** character indicates that there will be a parameter after the format String (i.e., the **change** variable). The **1.2f** indicates to JAVA that we want it to display the **change** as a floating point number with at least **1** digit before the decimal and exactly **2** digits after the decimal. The **,** character indicates that we would like it to automatically display commas in the money amount when necessary (e.g., $1,500,320.28). Apply this formatting to the total amount as well:

```
import java.util.Scanner;

public class ChangeCalculatorProgram2 {
    public static void main(String[] args) {
        double  price, total, payment, change;

        System.out.println("Enter product price:");
        price = new Scanner(System.in).nextFloat();

        total = price * 1.13;
        System.out.println("Total cost:$" + String.format("%,1.2f", total));

        System.out.println("Enter payment amount:");
        payment = new Scanner(System.in).nextFloat();

        change = payment - total;
        System.out.println("Change:$" + String.format("%,1.2f", change));
    }
}
```

Here is the resulting output for both test cases:

```
Enter product price:
35.99
Total cost:$40.67
Enter payment amount:
50
Change:$9.33
```

```
Enter product price:
8.85
Total cost:$10.00
Enter payment amount:
10
Change:$-0.00
```

It is a bit weird to see a value of **-0.00**, but that is a result of the calculation. Can you think of a way to adjust the **change** calculation of **payment - total** so that it eliminates the **-** sign ? Try it.

The **String.format()** can also be used to align text as well. For example, suppose that we wanted our program to display a receipt instead of just the change. How could we display a receipt in this format:

```
  Product Price     35.99
            Tax      4.68
------------------------
        Subtotal     40.67
Amount Tendered     50.00
========================
     Change Due      9.33
```

If you notice, the largest line of text is the "Amount Tendered" line which requires 15 characters. After that, the remaining spaces and money value take up 10 characters. We can therefore see that each line of the receipt takes up 25 characters. We can then use the following format string to print out a line of text:

```
System.out.println(String.format("%15s%10.2f", aString, aFloat));
```

Here, the **%15s** indicates that we want to display a string which we want to take up exactly **15** characters. The **%10.2f** then indicates that we want to display a float value with **2** decimal places that takes up exactly 10 characters in total (including the decimal character). Notice that we then pass in two parameters: which must be a **String** and a **float** value in that order (these would likely be some variables from our program). We can then adjust our program to use this new String format as follows …

```
import java.util.Scanner;

public class ChangeCalculatorProgram3 {
    public static void main(String[] args) {
        double    price, tax, total, payment, change;

        System.out.println("Enter product price:");
        price = new Scanner(System.in).nextFloat();

        System.out.println("Enter payment amount:");
        payment = new Scanner(System.in).nextFloat();

        tax = price * 0.13;
        total = price + tax;
        change = payment - total;

        System.out.println(String.format("%15s%10.2f","Product Price", price));
        System.out.println(String.format("%15s%10.2f","Tax", tax));
        System.out.println("------------------------");
        System.out.println(String.format("%15s%10.2f","Subtotal", total));
        System.out.println(String.format("%15s%10.2f","Amount Tendered", payment));
        System.out.println("========================");
        System.out.println(String.format("%15s%10.2f","Change Due", change));
    }
}
```

The result is the correct formatting that we wanted. Realize though that in the above code, we could have also left out the formatting for the 15 character strings by manually entering the necessary spaces:

```
System.out.println(String.format("  Product Price%10.2f", price));
System.out.println(String.format("            Tax%10.2f", tax));
System.out.println(              "-------------------------");
System.out.println(String.format("       Subtotal%10.2f", total));
System.out.println(String.format("Amount Tendered%10.2f", payment));
System.out.println(              "=========================");
System.out.println(String.format("     Change Due%10.2f", change));
```

However, the **String.format** function provides much more flexibility. For example, if we used **%-15S** instead of **%15s**, we would get a left justified result (due to the **-**) and capitalized letters (due to the capital **S**) as follows:

```
PRODUCT PRICE       34.99
TAX                  4.55
-------------------------
SUBTOTAL            39.54
AMOUNT TENDERED     50.00
=========================
CHANGE DUE          10.46
```

There are many more format options that you can experiment with. Just make sure that you supply the required number of parameters. That is, you need as many parameters as you have **%** signs in your format string.

For example, the following code will produce a **MissingFormatArgumentException** since one of the arguments (i.e., values) is missing (i.e., 4 % signs in the format string, but only 3 supplied values):

```
System.out.println(String.format("$%.2f + $%.2f + $%.2f = $%.2f", x, y, z));
```

Also, you should be careful not to miss-match types, otherwise an error may occur (i.e., **IllegalFormatConversionException**).

The next page shows a table of a few other format types that you may wish to use in the future. You are not responsible for knowing or memorizing anything in that table ... it is just for your own personal use.

Hopefully, you now feel confident enough to writing simple one-file JAVA programs to interact with the user, perform some computations and solve some relatively simple problems. It would be a VERY good idea to see if you can convert some of your simpler Processing/Python programs into JAVA.

Supplemental Information (Other String.format Flags)

There are a few other format types that may be used in the format string:

Type	Description of What it Displays	Example Output
%d	a general integer	4096
%x	an integer in lowercase hexadecimal	ff
%X	an integer in uppercase hexadecimal	FF
%o	an integer in octal	377
%f	a floating point number with a fixed number of spaces	83.43
%e	an exponential floating point number	7.869877e-03
%g	a general floating point number with a fixed number of significant digits	0.008
%s	a string as given	"Hello"
%S	a string in uppercase	"HELLO"
%n	a platform-independent line end	<CR><LF>
%b	a boolean in lowercase	true
%B	a boolean in uppercase	FALSE

There are also various format flags that can be added after the % sign:

Format Flag	Description of What It Does	Example Output
-	numbers are to be left justified	2378.348 followed by any necessary spaces
0	leading zeros should be shown	000244.87
+	plus sign should be shown if positive number	+67.34
(	enclose number in round brackets if negative	(439.67)
,	show decimal group separators	2,347,892.99

There are many options for specifying various formats including the formatting of Dates and Times, but they will not be discussed any further here. Please look at the java documentation.