Chapter 1

Programming Basics

What is in This Chapter ?

This first chapter gives a very brief introduction to what computer science is all about and what it means to program.   It then discusses how the **JAVA** programming language basically works as well as how to get a simple program up and running.   Also, since all programs should have some kind of input and output, we look here at how to write programs that *display* information and *get* information from the user. The chapter then discusses JAVA's **primitive data types** and how they are used as variable types. The **Math** class is also discussed, which allows for scientific calculations. The chapter then concludes with information on how to **format** your output nicely.

1.1 What is Computer Science ?

Computers are used just about everywhere in our society:

- Communications: internet, e-mail, cell phones
- Word Processing: typing/printing documents
- Business Applications: accounting, spreadsheets
- Entertainment: games, multimedia applications
- Database Management: police records, stock market
- Engineering Applications: scientific analysis, simulations
- Manufacturing: CAD/CAM, robotics, assembly
- ... many more ...
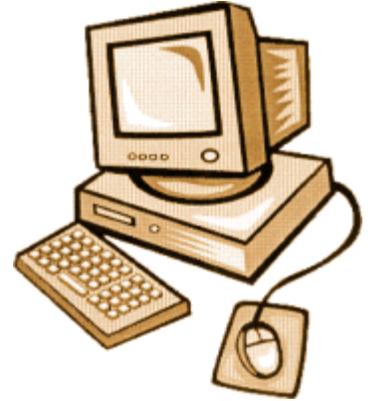
A computer is defined as follows (Wikipedia):

> A **computer** *is a programmable machine that receives input, stores and manipulates data, and provides output in a useful format.*

In regards to today's computers, the "*machine*" part of the computer is called the **hardware**, while the "*programmable*" part is called the **software**.

Since computers are used everywhere, you can get involved with computers from just about any field of study. However, there are specific fields that are more computer-related than others. For example. the fields of *electrical engineering* and *computer systems engineering* primarily focus on the design and manufacturing of computer *hardware*, while the fields of **software engineering** and **computer science** primarily focus on the design and implementation of *software*.

Software itself can be broken down into 3 main categories:

- **System Software:** is designed to operate the computer's hardware and to provide and maintain a platform for running applications. (e.g., Windows, MacOS, Linux, Unix, etc..)

- **Middleware:** is a set of services that allows multiple processes running on one or more machines to interact. Most often used to support and simplify complex distributed applications. It can also allow data contained in one database to be accessed through another. Middleware is sometimes called *plumbing* because it connects two applications and passes data between them. (e.g., web servers, application servers).

- **Application Software:** is designed to help the user perform one or more related specific tasks. Depending on the work for which it was designed, an application can manipulate text, numbers, graphics, or a combination of these elements. (e.g., office suites, web browsers, video games, media players, etc…)

The area of software design is huge.   In this course, we will investigate the basics of creating some simple application software.   If you continue your degree in computer science, you will take additional courses that touch upon the other areas of system software and middleware.

Software is usually written to fulfill some need that the general public, private industry or government needs. Ideally, software is meant to make it easier for the ***user*** (i.e., the person using the software) to accomplish some task, solve some problem or entertain him/herself. Regardless of the user's motivation for using the software, many problems will arise when trying to develop the software in a way that produces correct results, is efficient and robust, easy to use and visually appealing. That is where *computer science* comes in:

> **Computer science** *is the study of the theoretical foundations of information and computation, and of practical techniques for their implementation and application in computer systems* (Wikipedia).

So, computer science is all about taking in information and then performing some computations & analysis to solve a particular problem or produce a desired result, which depends on the application at hand.

Computer science is similar to mathematics in that both are used as a means of defining and solving some problem.   In fact, computer-based applications often use mathematical models as a basis for the manner in which they solve the problem at hand.

In mathematics, a solution is often expressed in terms of formulas and equations.   In computer science, the solution is expressed in terms of a *program*:

> A **program** *is a sequence of instructions that can be executed by a computer to solve some problem or perform a specified task.*

However, computers do not understand arbitrary instructions written in English, French, Spanish, Chinese, Arabic, Hebrew, etc..

Instead, **computers have their own languages** that they understand. Each of these languages is known as a programming language.

> A **programming language** *is an artificial language designed to automate the task of organizing and manipulating information, and to express problem solutions precisely.*

A programming language "boils down to" a set of words, rules and tools that are used to explain (or define) what you are trying to accomplish.   There are many different programming languages just as there are many different "spoken" languages.

Traditional programming languages were known as ***structural* programming** languages (e.g., C, Fortran, Pascal, Cobol, Basic).   Since the late 80's however, ***object-oriented* programming** languages have become more popular (e.g., JAVA, C++, C#)

There are also other types of programming languages such as *functional* programming languages and *logic* programming languages. According to the **Tiobe** index (i.e., a good site for ranking the popularity of programming languages), as of January 2015 the 10 most actively used programming languages were (in order of popularity):

C, **Java**, **Objective-C**, **C++**, **C#**, **PHP**, **Javascript**, **Python**, **Visual Basic.NET** and **Perl**.

When it comes to computers and software, there are many roles to play:

- ***System/Hardware Designers*** = design computers and related products.
- ***Manufacturers*** = actually build and assemble computers.
- ***Software Designers*** = design software to be used with the computers.
- ***Programmers*** = write the code to make the software work.
- ***End Users*** = buy and use the software when it is done.

We are going to play the role of the ***Programmer*** in this course by writing our own programs to solve some simple problems. In a way, we will also be playing the role of the ***End User*** when we test our programs.   Testing is an important part of programming to ensure that the program works properly for its intended purpose.

When thinking of jobs and careers, many people think that computer science covers anything related to computers (i.e., anything related to *Information Technology*).   However, computer science is not an area of study that pertains to IT support, repairing computers, nor installing and configuring networks.   Nor does it have anything to do with simply using a computer such as doing word-processing, browsing the web or playing games.   The focus of computer science is on understanding what goes on behind the software and how software/programs can be made to operate more efficiently.

The **Computer Sciences Accreditation Board** (CSAB) identifies four general areas that it considers crucial to the discipline of computer science:

- *theory of computation*
  - investigates how specific computational problems can be solved efficiently

- *algorithms and data structures*
  - investigates efficient ways of storing, organizing and using data

- *programming methodology and languages*
  - investigates different approaches to describing and expressing problem solutions

- *computer elements and architecture*
  - investigates the design and operation of computer systems

However, in addition, they also identify other important fields of computer science:

- software engineering
- artificial intelligence
- computer networking & communication
- database systems
- parallel computation

- distributed computation
- computer-human interaction
- computer graphics
- operating systems
- numerical & symbolic computation

There are aspects of each of the above fields can fall under the general areas mentioned previously.   For example, within the field of **database systems** you can work on theoretical computations, algorithms & data structures, and programming methodology.

As you continue your studies in computer science, you will be able to specialize in one or more of these areas that interest you.   This course, however, is meant to be an introduction to programming computers with an emphasis on problem solving.

This is your first programming course here in the School of Computer Science at Carleton.   You have some more *core* programming courses coming up after this one.   Here is a break-down of how this course fits in with your first 2 years of required programming courses:

This is your first programming course here in the School of Computer Science at Carleton.   You have some more *core* programming courses coming up after this one, as shown here.

After this course is over, you *should* understand how to write computer programs to solve problems.   We will discuss the basics of programming including variables, loops, conditional statements, functions & procedures, data types, arrays, recursion, objects and classes.   In the winter term, you will take COMP1406 which is like a continuation of this course but deals with object-oriented programming concepts as well as more advanced use of data structures and recursion as well as aspects of building applications with user interfaces. Together, these two courses will give you a solid programming background in JAVA and you will be able to learn other computer languages easily afterwards … since they all have common features.   If you want to do well in this course, attend all lectures and tutorials and do your assignments.

1st Year

COMP 1405
(Java)

You are here.

COMP 1406
(Java)

2nd Year

COMP 2401
(C)

COMP 2404
(C++)
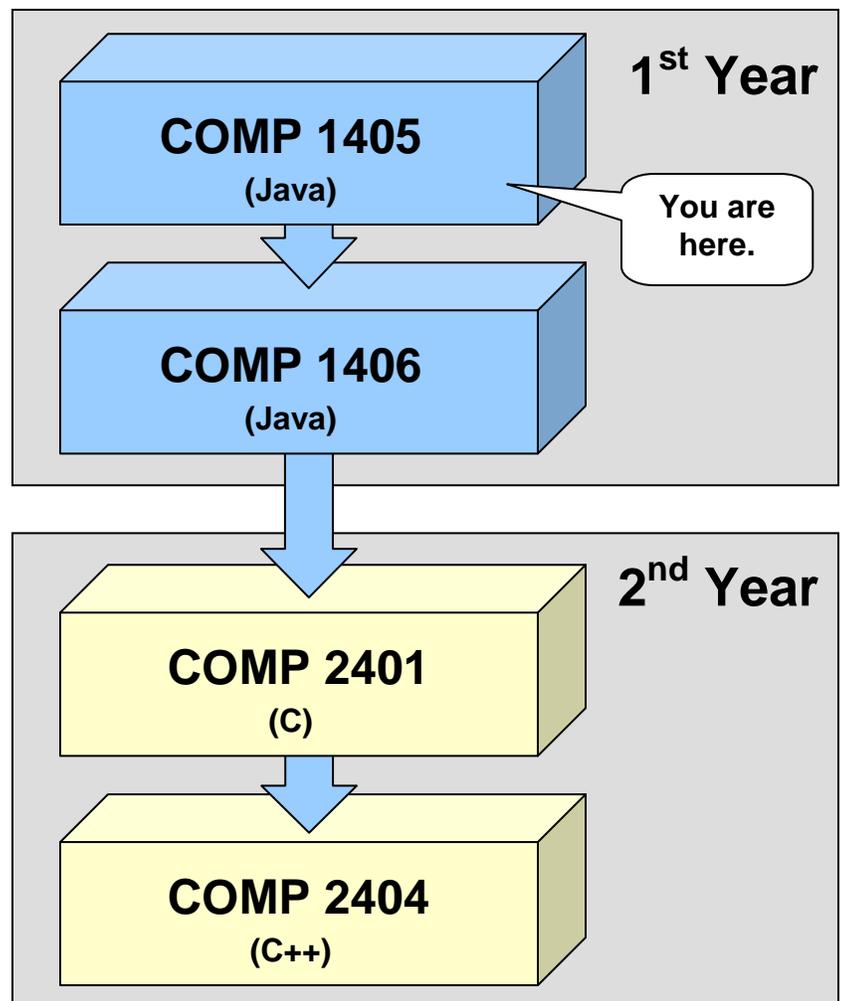
1.2 **The JAVA Programming Language**

JAVA is a very popular object-oriented programming language from SUN Microsystems. It 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)
- J2ME (for wireless communications such as cell phones, PDAs)

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/java 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 (e.g., web)
    - uses RMI (Remote Method Invocation) API

- **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 COMP2401)
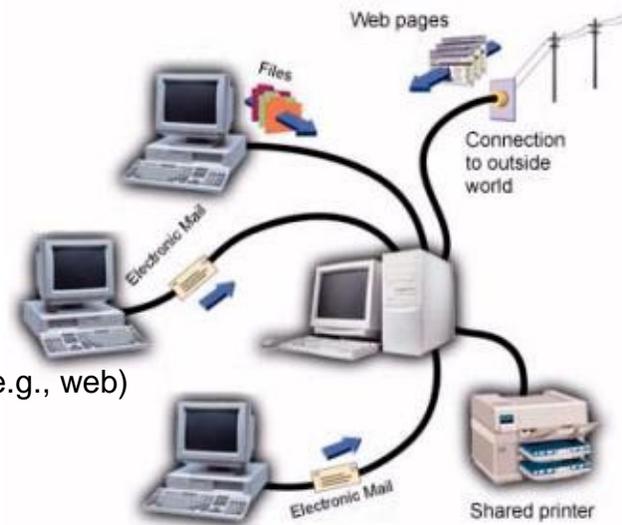
The JAVA programming language itself (i.e., the SDK that you download from SUN) actually consists of many program pieces (or object class definitions) which are organized in groups called **packages** (i.e., similar to the concept of _**libraries**_ in other languages) which we can use in our own programs.

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?

We will be using the latest version of JAVA (see course outline) which you can download from the Oracle website (www.oracle.com/java) if you are working at home.

When you download and install the latest **JAVA SDK** (i.e., JAVA Software Development Kit), 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 Oracle "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**

1.3 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.
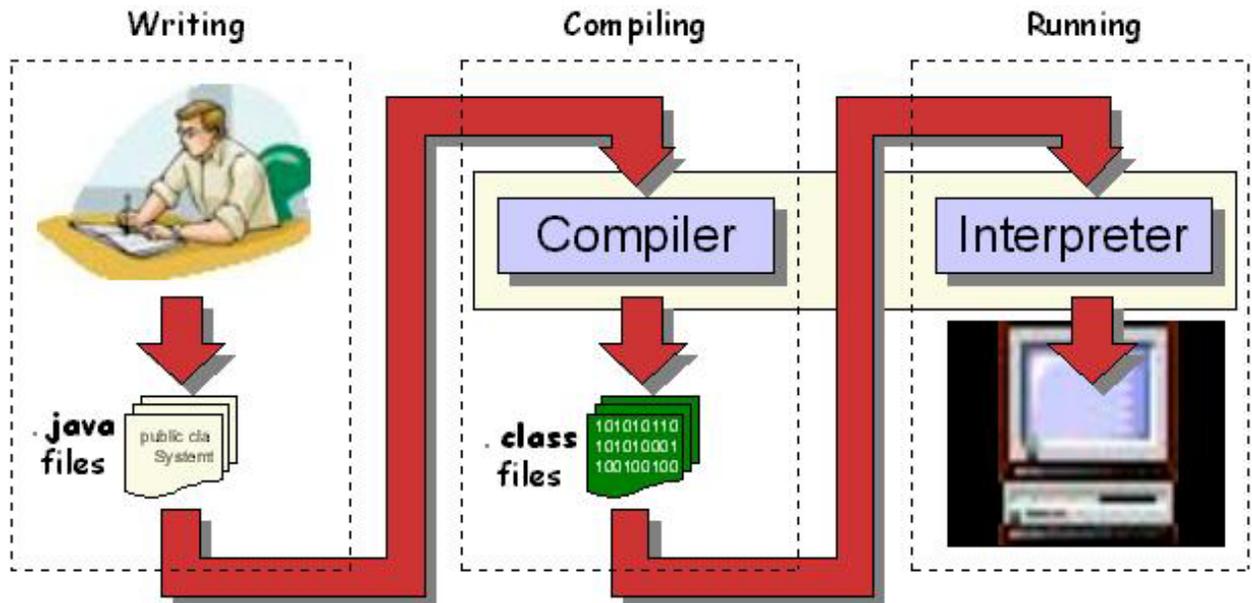
The java *compiler*:
- prepares your program for running
- produces a **.class** file containing **_byte-codes_** (which is a program that is ready to run).

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.

As mentioned, the JAVA language consists of various class libraries that you can make use of. All of JAVA's classes are arranged in **_packages_**.   There are MANY standard packages in JAVA, each with many classes.

Here are just some of the standard packages that you will likely use in this course:

`java.lang`	Basic classes and interfaces required by many JAVA programs. It is **automatically imported** into all programs.
`java.util`	Utility classes and interfaces such as date/time manipulations, random numbers, string manipulation, collections ...
`java.io`	Classes that enable programs to input and output data.
`java.text`	Classes and interfaces for manipulating numbers, dates, characters and strings. Provides internationalization capabilities as well.

When you want to make use of some of these classes, you will use the **import** keyword to tell JAVA that you want to use a class:

```
import <packageName>.*;
```

Basically, the **import** statement is used to tell the compiler which package (i.e., directory) the class files are sitting in.   You can always replace the * by a class name (where the class name is in the package) so that the readers of your code are more clear on which classes you are actually using.   Keep in mind though that the import statement ***does not load*** any classes, it merely instructs the compiler where to find them when you run your code.

Our First Program

The first step in using any new programming language is to understand how to write/compile and run 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.

All of your programs will consist of one or more files called ***classes***.   That is, each time you want to make a program, you need to define a ***class***.

Here is the 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 <u>ALL</u> 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).

- 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 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.

- 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.   The text between the double quotes is called a *String*. In fact, we could replace the **Hello World** text with any characters that we want displayed.   Notice also that there is a semicolon character (**;**) at the end of the line.   All JAVA statements (i.e., lines of code) will end with a **;** character.

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 <u>ALWAYS</u> start with a capital letter.   Also, your code <u>MUST</u> 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 <u>ALWAYS</u> line up ALL of your brackets using the **Tab** key on the keyboard.

Where do we write this code ?   In the **DrJava** IDE.   **DrJava** has a main window area into which we write the code and another area at the bottom where we view the results:

Here are the steps to get your program ready to run:
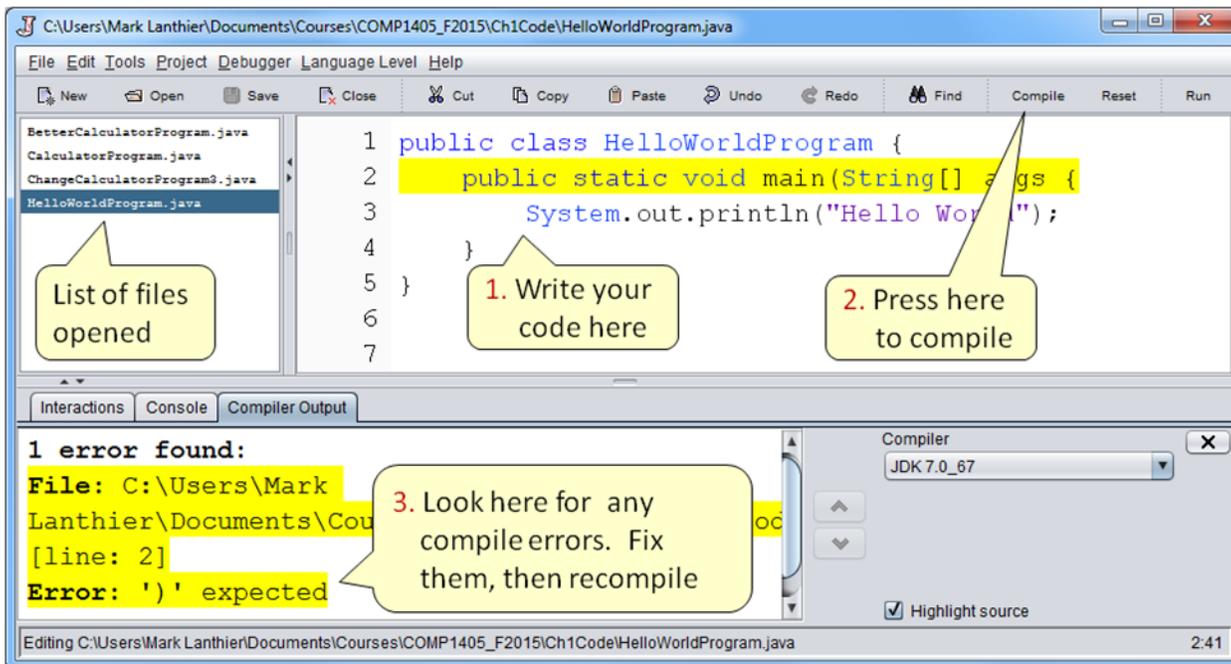
Once your code compiles without error ...

Then you can then run it and view the output:
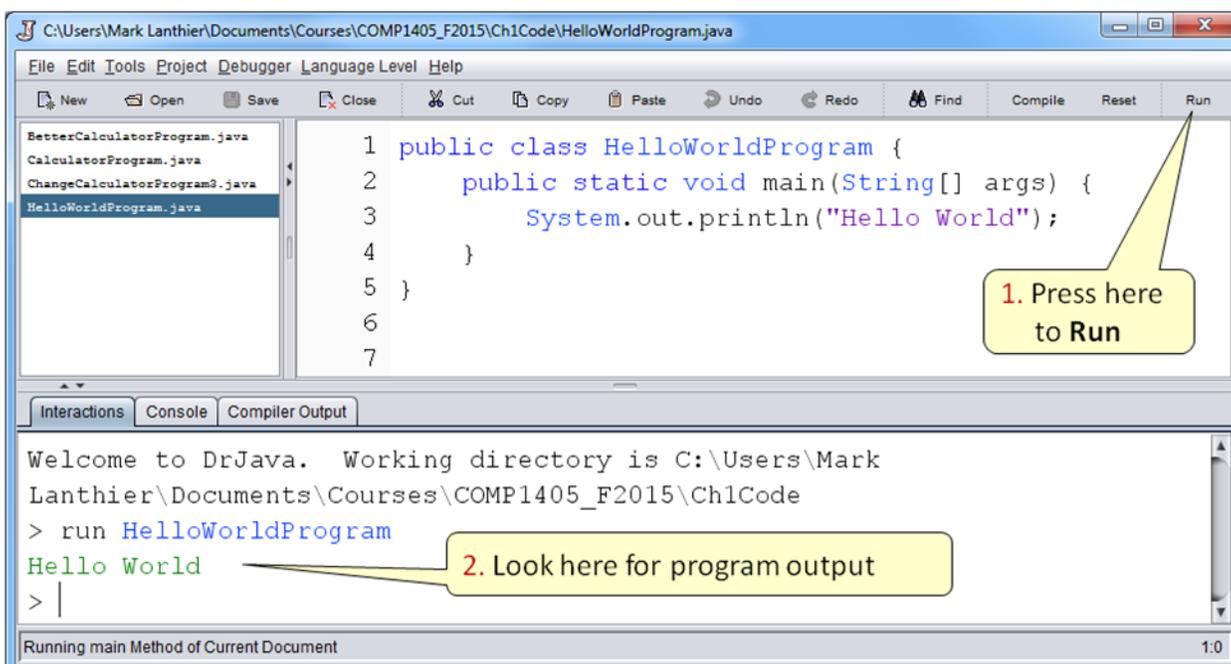
1.4 Displaying Information

We have seen in our first program how to use the **System.out.println()** statement to output a simple line of text characters to the screen. However, this JAVA statement can do much more. For example, it can also output *results* of computations. In this section we will look at a few more examples of what can be displayed on the screen.

Here is another program which represents a *calculator* that can find the average of three numbers (e.g., 34, 89 and 17) and display the answer on the console window:

```java
public class CalculatorProgram {
    public static void main(String[] args) {
        // This code computes a simple calculation
        System.out.print("The average of 34, 89 and 17 is ");
        System.out.println((34 + 89 + 17) / 3.0);
    }
}
```

There are some points of interest regarding the code:

- In addition to displaying text characters, the **System.out.println** can display the "results" of mathematical computations.

- The code in green (i.e., following the **//** characters) is called a ***comment***.   JAVA ignores this when compiling.   You can place comments anywhere in your code to provide an explanation of what your code is doing.   This helps later on when you look at your code at a future date ... because we all tend to forget what we did in the past.

    - Generally, you should use **//** when you have a single line to comment (i.e., everything after the **//** characters on that line is ignored).
    - If you want to have a multiple line comment, you can alternatively begin the comment with **/*** characters and end it with ***/** characters. For example:

      ```java
      /* This is a multiple line comment
         because it appears on more
         than one line in the program. */
      ```

      Of course, you can still use the **//** characters instead 3 times if you want:

      ```java
      // This is a multiple line comment
      // because it appears on more
      // than one line in the program.
      ```

- • The first line uses **print** while the second line uses **println**.   When using just **print,** the next text to be printed will be immediately to the right of this text. When using **println**, a line feed and carriage return is printed, which means that the text to follow will appear at the beginning of the next line of the console (i.e., output window).

Here is the output when the **CalculatorProgram** is run:

```
The average of 34, 89 and 17 is 46.666666666666664
```

Of course, this program always computes the same average using the same 3 numbers, but in the next section, we will look at how to get *different* numbers from the user each time we run the code. Notice as well that the calculations are not perfectly accurate … they are off a little after 15 decimal places.

Can you tell how the output of the following piece of code will be formatted ?

```
System.out.print("My name is ");
System.out.println("Mark.");
System.out.println("These strings " + "are" + " joined.");
System.out.print("Numbers can be appended ... see: " + 54.342);
System.out.println(" and even characters: " + 'A' + 'B' + 'C');
System.out.println();
System.out.println("The line above was left blank.");
System.out.println("Now leave 4 blank lines at the end. \n\n\n\n");
System.out.println("Count the blanks above.");
```

Here is the output:

```
My name is Mark.
These strings are joined.
Numbers can be appended ... see: 54.342 and even characters: ABC

The line above was left blank.
Now leave 4 blank lines at the end.




Count the blanks above.
```

Notice that we can use the **+** to join:

- • two strings before display them
- • numbers or characters (defined between single quotes **' '**) to the end of a String.

Also notice that:

- when the brackets **()** are left empty on a **println()**, then a blank line is printed.
- we can use some **\n** characters at the end of the string to leave blank lines.

Displaying results in a *window*

The output of our programs above was displayed in the bottom part of the DrJava window (called the *console*). However, we can also have our program results appear in a nice little window that pops up on the screen.   The code behind making a window can be a little confusing at this point in the course, so we will not make our own.   Instead, JAVA has some pre-defined windows called **JOptionPanes** which will allow us to display some simple test results.

The simplest way to do this is to use one of the standard *dialog boxes* in JAVA. We can use something called a **JOptionPane** which is in the **javax.swing** package.

Consider for example, our **CalculatorProgram** that we wrote earlier.   We can bring up a little window with our answer in it as follows:
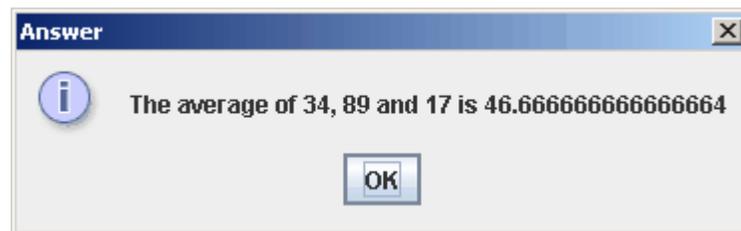
The window would come up, wait for us to press the **OK** button and then close.   Here is the code that does this:

```java
import javax.swing.JOptionPane;

public class WindowCalculatorProgram {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null,
                    "The average of 34, 89 and 17 is " + (34+89+17)/3.0);
    }
}
```

Basically, it is a one-line-program again.   The **JOptionPane.showMessageDialog(null, …);** "command" is a pre-defined JAVA function that brings up the window with the text that we choose (just replace the **...** characters with your text).   **JOptionPane** is actually the JAVA class that does this for us. Remember that since we are using a pre-defined JAVA class, we need to tell the JAVA compiler where to find it.   That is why we write the following statement at the top of our program:

```
import javax.swing.JOptionPane;
```

Another way to understand the **import** statement is to think of it as something you write at the top of your program to tell the compiler which "tools" you will be using in your program.

The **JOptionPane** class has a *function* (officially known as a *method*) called **showMessageDialog** which contains code for displaying the window.   This method requires you to supply 2 pieces of information called *parameters*. Each of these parameters is separated by a comma character and is shown on a separate line in the code to make things clearer:

1. The 1st parameter is **null**.   We will discuss this later.
2. The 2nd parameter is the information that you want to display in the middle of the window. It can be any code, but is often a String.

Experiment with the example above by trying to display various things in the window.

Normally, **JOptionPanes** are meant to have a single line of text. However, if you want to have multiple lines of text, you can do this by appending a **\n** character between the lines that you want. This will tell JAVA to go to the next line before continuing.   For example, if we add some **\n** characters to our String as shown below, notice what the window will look like:

```
JOptionPane.showMessageDialog(null,
          "The average of \n34, 89 and 17\nis\n\n" + (34 + 89 + 17) / 3.0);
```

Here is the window that is produced:
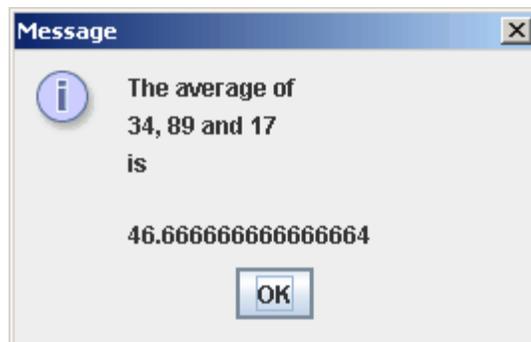
Of course, the appearance is not as pleasant because everything is aligned to the left.   As is, there is no way to change this alignment. Nevertheless, it is sometimes nice to be able to display text in our own windows as opposed to simply in the console window.

In the follow-up course to this one (COMP1406), you will learn how to create your own windows and arrange everything as you want.

Supplemental Information (Other MessageDialogs)

There are variations for the **showMessageDialog** method that allow 2 more parameters so that you can change the title on the window as well as add a picture. The format is:

```
JOptionPane.showMessageDialog(null,
    "The average of 34, 89 and 17 is " + (34 + 89 + 17) / 3.0,
    "Answer",
    JOptionPane.PLAIN_MESSAGE);
```

The 3rd parameter here is the title for the window, in this case "Answer".   The 4th parameter is the type of window to open.   Here we say **JOptionPane.PLAIN_MESSAGE**, but there are other options which will bring up the window but will also display a little picture (called an *Icon*) that allows you to distinguish between different kinds of messages.   Below is a table of the other options and their icons.   In fact, you can look at the JAVA documentation and see that you can also use your own pictures/icons.
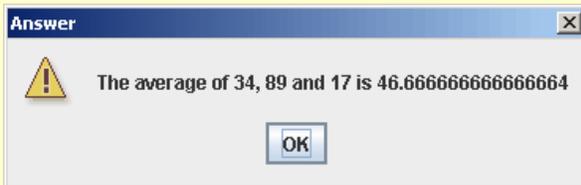
JOptionPane.WARNING_MESSAGE
Use this when you want to warn the user about something in the program.

JOptionPane.ERROR_MESSAGE
Use this when you want to tell the user that an error has occurred in the program.
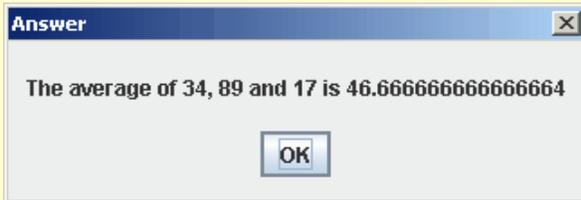
JOptionPane.PLAIN_MESSAGE
Use this when you do not want a picture.
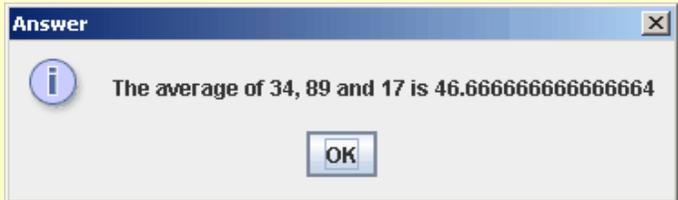
JOptionPane.INFORMATION_MESSAGE
Use this when you want to tell the user something in the program.

1.5 Getting User Input

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.   Java has a pre-defined "tool" which allows you to get input from the keyboard in a simple manner. The tool is in a class called **Scanner** and it is available if you import the **java.util** package.

So, to get input from the user, we need to create a new **Scanner** *object* for the **System** console.   We will talk much more at a later time about creating new objects, but for now, 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 the user for his/her name and then says hello to him/her:

```java
import java.util.Scanner;

public class GreetingProgram {
    public static void main(String[] args) {
        System.out.println("What is your name ?");
        System.out.println("Hello, " + new Scanner(System.in).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:

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

For example, consider this modified calculator program that finds the average of three numbers entered by the user:

```java
import java.util.Scanner;

public class BetterCalculatorProgram {
    public static void main(String[] args) {
        System.out.println("Enter three numbers:");
        System.out.println("The average of these numbers is " +
                            (new Scanner(System.in).nextInt() +
                             new Scanner(System.in).nextInt() +
                             new Scanner(System.in).nextInt()) / 3.0);

    }
}
```

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

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

Supplemental Information (Bug)

In some versions of JCreator, there was a *bug* when getting keyboard input from the output window.   The above code for example, when using the "**Capture output**" feature in JCreator (under the **RunApplication** tool option) will give the value of the first number entered (which is 34 here).   If this happens to you, you can avoid this problem by disabling (i.e., uncheck) the "**Capture output**" feature when running programs that require keyboard input.

Of course, we can enter any numbers.   Here is the output from entering 10, 20 and 50:

```
Enter three numbers:
10
20
50
The average of these numbers is 26.666666666666668
```

As we will see in the next section, we do not need to make 3 **Scanner** objects, we can actually use the same **Scanner** each time.   There is much more we can learn about the **Scanner** class.   It allows for quite a bit of flexibility in reading input.

For example, if we enter 10, 20 and then some junk characters ... an error will occur as follows:

```
Enter three numbers:
10
20
junk
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at BetterCalculatorProgram.main(BetterCalculatorProgram.java:6)
```

Woops!   That's not very nice.   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.

Getting input from a *window*

Getting input directly from the keyboard into a console is an obsolete task these days. Most software has some kind of user interface that allows the user to enter text using dialog boxes, text fields, sliders, list boxes etc...   In JAVA, we can use also **JOptionPane** to get input from the user.   If we would like to ask the user for a simple piece of text, such as his/her name, we can use the following line: `JOptionPane.showInputDialog("Please input your name");`

This code will bring up the following window which will let us enter the name:

So, the pre-defined **JOptionPane** class has a **showInputDialog(…)** method that brings up a window and waits for us to enter data.   The one parameter to that method allows us to give instructions to the user in the form of a sentence.   Consider now changing our **GreetingProgram** to use this new window instead:

```java
import javax.swing.JOptionPane;


public class WindowGreetingProgram {
    public static void main(String[] args) {
        System.out.println("Hello, " +
                    JOptionPane.showInputDialog("What is your name ?"));
    }
}
```

Notice that the output from this program in the **System.out** console only displays the result:

```
   Hello, Mark
```

We can actually combine this with the **showMessageDialog** method as follows:

```java
import javax.swing.JOptionPane;

public class WindowGreetingProgram2 {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Hello, " +
                    JOptionPane.showInputDialog("What is your name ?"));
    }
}
```

Then we end up with windows for input and for output … with nothing printed to the console:

There are other types of input windows.   For example, we can bring up a window that asks the user a YES/NO question as follows:

```java
JOptionPane.showConfirmDialog(null, "Are you sure you want to quit ?");
```

This method will bring up the following window:

We will see later that we can find out which of the buttons the user pressed (i.e., Yes, No or Cancel) and then respond accordingly in our program.   We will not discuss any other types of input windows at this point because they require you to understand more about JAVA.

So, you should now have a solid understanding of how to display information as well as get information from the user.   We will now continue to get a better understanding of the JAVA programming language.

1.6 Primitive Data Types and Variables

Up until this point, we have done some simple programs that perform some simple calculations.   In general, a typical computer is only able to compute one piece of information at a time. However, for big problems, there may be a lot of computations, number crunching, data organization, etc…   So, in order to write some more meaningful programs, we need to understand how to store information.

> *A* *variable* *is a location in the computer's memory that stores a single piece of data.*

A program may use many variables to store intermediate results.   For example, recall our example for averaging 3 numbers entered by the user:

```java
import java.util.Scanner;

public class BetterCalculatorProgram {
    public static void main(String[] args) {
        System.out.println("Enter three numbers:");
        System.out.println("The average of these numbers is " +
                            (new Scanner(System.in).nextInt() +
                             new Scanner(System.in).nextInt() +
                             new Scanner(System.in).nextInt()) / 3.0);
    }
}
```

In this example, there was no need to store the data that the user entered because it was used right away in the program.   However, what if we then wanted to find the maximum, the minimum and perhaps the sales tax on the total ?   We would need to store these numbers somewhere so that we can do multiple computations on the same numbers without having to re-enter them again.

Sometimes students have a hard time knowing when a variable is needed.   Just remember that a variable is just *storage space*.   A typical computer program can only do one thing at a time because it has a single processor. Imagine, for example, trying to pour yourself a glass of orange juice with just one hand.   You would likely do things in this order:

1. get a glass
2. put it down on the countertop
3. get the orange juice carton
4. put it down on the countertop
5. open the orange juice carton
6. pour the orange juice into the glass
7. put down the orange juice carton
8. pick up the glass and drink it

You can see that the counter itself represents the "space" that you can store things onto and pick them back up later.   The countertop is required in order for you to complete your one-handed task. Similarly variables are usually required in order to complete your program.

The countertop is like the computer's memory and each object placed on it takes up space on the countertop. All information in the computer is actually stored in the electronics as voltages … high and low voltages that can be thought of as billions of 1's and 0's that have some kind of meaning to them.   That is, all user information (whether it is a name, phone number, picture, email, database, game, etc..) is stored as 1's and 0's which we call *bits*.

As humans, we have a hard time working at such a low level.   We do better working with things like numbers, characters and real-world objects.   So, rather than work with single bits, we group these bits into more abstract or higher-level packages.

A group of 8 bits is known as a *byte*. A byte can represent 256 combinations of 1's and 0's. That is, if we think of each bit as being a switch which is either on or off, we can flip the 8 switches in 256 unique combinations.   This allows a single byte to store a number from **0 to 255**.
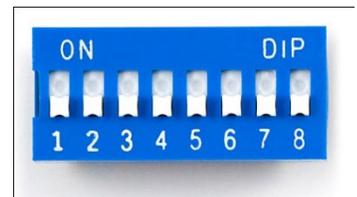
When two bytes are required to represent a number, the pair of bytes is called a *word*. A *word* can store a number in the range from **0 to 65535**.   We can continue to group bytes together to store even larger integer numbers.

So, the *bit* and the *byte* are the two most primitive forms that a computer uses for number representation. Most computers will use the term *boolean* to represent a 0 or 1, but instead of saying "0" or "1" the terms "*false*" and "*true*" are used.
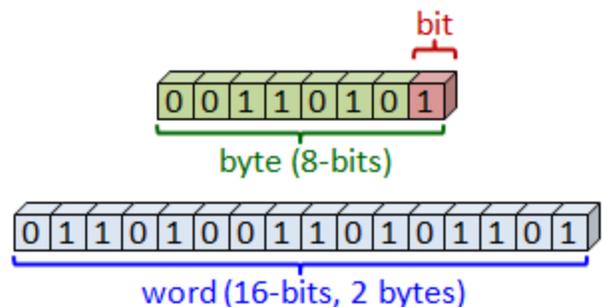
Bytes can also be used to represent letters, digits, punctuation, etc. which are called *characters*.   How so ? Well, back in 1968 it was decided that computers conform to a numbering standard called *ASCII* (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange).   That is, each combination of numbers in the range from 0 through 127 was mapped to (i.e., corresponds to) a particular keyboard character.   Here is the ASCII table (provided as a reference only … DO NOT try to memorize it):

ASCII value	Character(s)	ASCII value	Character(s)
0	null	48-57	`0123456789`
1-31	various special characters	58-64	`:;⇔?@`
10	line feed	65-90	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
13	carriage return	91-96	`[\]^_``
32	space	97-122	`abcdefghijklmnopqrstuvwxyz`
33-47	`!"#$%&'()*+,-./`	123-127	`{|}~□`

So, for example, the letter "**A**" corresponds to number **65** in the ACSII table which is number **01000001** in binary bits (we will not discuss bit representation any further in this course). There are also versions of extended ACSII tables covering the numbers from 128 to 255.   In addition, since computers began to be used internationally, 256 combinations were not enough to represent the letters of various international languages.   Therefore, a new standard called *Unicode* has been developed (and continues to be expanded) to account for the other characters.   However, a single byte is no longer sufficient to represent the character … two or more bytes are required.

In addition, we can actually use bytes to represent real numbers (also called floating-point numbers) such as **3.14159265**.   Also, by making assumptions on one particular bit in a byte (i.e., the *most significant bit*, also called the *sign bit*), we can allow the numbers to be either negative or positive.   There are many details regarding number representation, but we will not discuss them further in this course.

The point is that *bits* are grouped to form *bytes* (or characters) which are also grouped to form larger numbers.   Ultimately, this leads to what are known as *primitive data types* that are used in most programming languages.   Here are the four basic primitives that are available in most programming languages (although the names may differ in each language):

- booleans – **true** or **false**
- integers – positive or negative whole numbers
- floating-point numbers – positive or negative real numbers with decimal places
- characters – letters, digits, punctuations or some other keyboard character

These are called *primitive* because they are the most **basic** types of data that we can store on the computer.   Some languages will further distinguish between various types of integers or floats.

For example, the following are the four official primitive data types in JAVA that can represent integers of various sizes:

Type	Bytes Used	Can Store an Integer Within this Range
byte	1	-128 **to** +127
short	2	-32,768 **to** +32,767
int	4	-2,147,483,648 **to** +2,147,483,647
long	8	-9,223,372,036,854,775,808 **to** +9,223,372,036,854,775,807

Notice that the various types take up a different amount of memory space.

Similarly, there are two official primitive data types in JAVA to store floating-point numbers:

Type	Bytes Used	Can Store a Real Number Within this Range
float	4	-10^{38} **to** $+10^{38}$
double	8	-10^{308} **to** $+10^{308}$

Regardless of how we group the bytes, all information/data can be represented through the 4 basic primitives of boolean, integer, floating point numbers and characters.

The final two primitive types are for storing single characters and booleans:

Type	Bytes Used	Stores Single Character
char	2	Represented by single quotes (e.g., 'a' 'B' 'c' '$' '>')
Type	Bytes Used	Stores Boolean Value
boolean	1	a value of either **true** or **false**

So those are the **8** primitive data types that JAVA offers.   You may have noticed that **Strings** are not listed there.   That is because **Strings** are actually *objects* which are made up of multiple **char** primitives.   That is, **Strings** are nothing more than a group of characters.

So why are we talking about this ?   Well, when programming, some languages (like JAVA) ***force you to specify the types*** for all of your variables.   That means, for every variable that you create, you must indicate its <u>type</u> as well as its <u>name</u>.

In JAVA, for example, in order to use a variable to store a primitive kind of value (e.g., a boolean, integer, floating-point number or character), you must specify in your program the **type** followed by the **name** of the variable.   Each variable must be given a **unique** name so that it can be identified later.   That way, JAVA knows exactly which object you want from the countertop. Just imagine that you are asking a robot to get the object off of the counter ... you need to tell it which object you want ... and you use the object's unique name to make that decision.
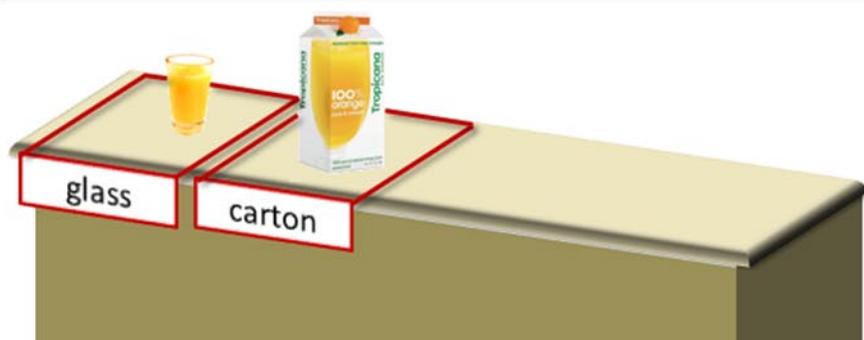
Here are examples of variables declared with their type on the left and name on the right:

```
boolean      hungry;
int          days;
byte         age;
short        years;
long         seconds;
char         gender;
float        amount;
double       weight;
```

The above examples show all 8 primitive types that you may use in JAVA programs.   Note that these will differ from language to language.   Notice as well that there is a **;** character after each line, as with any line of programming code.

Each line above is responsible for *declaring* a variable.   That means that a space is reserved in the computer's memory (with the given label) that can hold a value of the given type.

Declaring a variable, DOES NOT assign it any value, it only reserves space for the variable. In JAVA, after you declare a variable, you MUST give it a value *before* you use it.   For example, suppose that tried to do this within one of your programs:

```java
int  days;

System.out.println(days);  // prints out the days variable's value
```

You would get the following error, preventing your program from running:

```
The local variable days may not have been initialized
```

A note about variable names … make sure to pick **meaningful** names that are not too long !! The name must be unique and it is case-sensitive (i.e., **Hello** and **hello** would not be considered the same).

Variable names may contain only letters, digits and the **'_'** character (i.e., no spaces in the name).   As standard convention, multiple word names should have every word capitalized (except the first).

Here are some good examples of variable names:

- count
- average
- insuranceRate
- timeOfDay
- poundsPerSquareInch
- aString
- latestAccountNumber
- weightInKilograms

We use the term *assigning* to represent the idea of "giving a value" to a variable.   In JAVA, the *assignment operator* is the **=** sign.   So, we use **=** to put a value into a variable.

Here are a few example of how we can do this with some of the variables that we declared earlier:

```java
hungry = true;
days = 15;
gender = 'M';
amount = 21.3f;       // floats must have an 'f' after them
weight = 165.23;
```

Something VERY important to remember when learning to program is that the value of the variable **must be the same type** of object (or primitive) **as the variable's type** that was specified when you declared it earlier.   So for example, in the following table, make sure that you understand why the examples on the left are wrong, while the right examples are correct:

❌	`int days;` `days = 10.2789;`	✅	`int days;` `days = 10;`
❌	`boolean hungry;` `hungry = 'y';`	✅	`boolean hungry;` `hungry = false;`
❌	`char sex;` `sex = "F";`	✅	`char sex;` `sex = 'F';`
❌	`float weight;` `weight = 154.2;`	✅	`float weight;` `weight = 154.2f;`

To help cut down the number of lines of code in our program, we are allowed to both *declare* and *assign* a value to our variables all on one line.   So, from our earlier examples, we can do the following:

```
boolean    hungry = true;
int        days = 15;
char       gender = 'M';
float      amount = 21.3f;
double     weight = 165.23;
```

A variable may be ***declared*** only once in the program, but we may ***assign*** a value to it multiple times.

Can you determine the output of this piece of code:

```
int   days;

days = 43;
System.out.println(days);   // prints out 43

days = 15;
System.out.println(days);   // prints out 15
```

So, variables can be ***re-assigned*** a value, but cannot be ***declared*** again.   Therefore, the following code will NOT compile:

```
int days = 365;
System.out.println(days);
int days = 7;                    // cannot declare days a second time
System.out.println(days);
```

Here are some more pieces of code. Do you know what the output is ?

```java
int x;
int y;
x = 34;
y = 23;
System.out.println(x + y);
```

Here is a similar example.   Notice in JAVA that we are allowed to declare multiple variables of the same type on the same line, each separated by a **','**:

```java
int x, y;
x = 34;
y = x;
System.out.println(x + y);
```

Here is another one:

```java
int x, y, z;

x = 3*2*1;
y = x + x;
z = x;
System.out.println(z);
```

Note that even though we use **x** a few times, it does not change its value.

Here is one that is a little more interesting:

```java
int    total;
float  average;

total = 12 + 25 + 36 + 15;
average = total / 4;
System.out.print("The average is ");
System.out.println(average);
```

Here is the output:

```
The average is 22.0
```

Keep in mind that each time we call **System.out.print()**, any further information displayed will appear on the same line, so it is important to have the extra space character at the end of the string above, otherwise the result would be crowded close to the text like this:

```
The average is22.0
```

We can also combine the two print statements into one line as follows:

```java
System.out.println("The average is " + average);
```

This code will append the current value of the **average** variable to the string.

Notice in our code so far we have used numbers directly (e.g., 34, 89, 34.52) as well as characters (e.g., 'a', 'M', '\n'). These are called *literal values* (a.k.a. *literals* or *constants*) because the value is *literally* what you see when you read it in the program.

In JAVA, when you use a literal real number constant such as **34.685**, JAVA assumes that you want this number to be used as a **double**. If you just want it to be a **float** (which has half the precision), then you must supply an **f** character after the number as follows: **34.685f** … otherwise … JAVA may give you a compile error. Similarly, **long** numbers must have an **L** after them to distinguish them from **ints**.

There are some special pre-defined characters that have special meanings. They are actually specified as 2 characters ... the backslash being the first.   Here is a list of just a few of them:

- **'\n'**   (newline)
- **'\b'**   (backspace)
- **'\''**   (single quote)
- **'\t'**   (tab)
- **'\\'**   (backslash)
- **'\"'**   (double quote)

It is likely that sooner or later you will need to use these special characters in a **String**.

If you have a value that will remain constant throughout your program you can use the keywords *static final* (implying that it has its final value and will not change again) before the variable's type.

> *A constant is a single piece of data that does not change throughout the algorithm*

In this case, you must assign the value to the constant when it is declared:

```
static final int      DAYS = 365;
static final float    INTEREST_RATE = 4.923f;
static final double   PI = 3.1415965;
```

Normally, constants use uppercase letters with underscores (i.e., _) separating words.

Here is a simple test program that prints out some primitive values:

```java
public class PrimitiveTestProgram {
    static final int      DAYS = 365;
    static final float    INTEREST_RATE = 4.923f;


   public static void main(String[] args) {
        System.out.println(239);                    // an integer
        System.out.println(823100267876L);        // a long
        System.out.println(3.141592653589793);  // a double
        System.out.println(3.141592653589793f); // a float

        System.out.println('A');  // a letter character
        System.out.println('5');  // a digit character
        System.out.println(' ');  // the space character
        System.out.println('\"'); // the double-quote character
        System.out.println('\n'); // the blank-line character

        System.out.println(true);  // the boolean value true
        System.out.println(false); // the boolean value false

        System.out.println(DAYS);                   // a constant
        System.out.println(INTEREST_RATE/12);  // a constant in a formula
    }
}
```

Here is the resulting output:

```
239
823100267876
3.141592653589793
3.1415927
A
5

"


true
false
365
0.41024998
```

Do you remember doing **new Scanner(System.in).nextLine()** when getting user input ?
Well, 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()** method available.   The **next()** method 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.

Example:

A team of **n** students work *together* painting houses for the summer. For each house they paint they get **$256.00**. Assume that the students work for **4** months of summer and their expenses (as a team) are **$152.00** per month.   Write a program that asks the user for the number of students on the team and then computes and displays the total number of houses that they must paint (as a team) for each student to have one thousand dollars at the end of the summer.

To begin, we will need to ask the user how many students are on the team:

```java
public class PaintHousesProgram {
    public static void main(String[] args) {
        System.out.println("How many students are on the team ? ");
    }
}
```

We will then need to ask the user for the number and store this in a variable so that we can compute an answer afterwards.   We will need an integer variable to store the number.   It is often a good idea to declare input-related variables at the top of your program, and then assign them a value in your program as needed.

```java
import java.util.Scanner;

public class PaintHousesProgram {
    public static void main(String[] args) {
        int     n;

        // Get the user input
        System.out.println("How many students are on the team ? ");
        n = new Scanner(System.in).nextInt();
    }
}
```

Now, we need to compute the answer and display it:

```
import java.util.Scanner;

public class PaintHousesProgram {
    public static void main(String[] args) {
        int     n;

        // Get the user input
        System.out.println("How many students are on the team ? ");
        n = new Scanner(System.in).nextInt();

        // Compute the answer
        int goalAmount = (n * 1000) + (4 * 152);
        int houses = goalAmount / 256;

        // Display the answer
        System.out.println(houses + " houses must be painted.");
    }
}
```

Example:

There are **n** kids in a room. As it turns out, some kids have socks on (with or without shoes), some kids are wearing shoes (with or without socks), and some kids are wearing both socks & shoes. See if you can come up with a program that asks the user for the information about the kids and then computes how many kids are barefoot (i.e., no socks, nor shoes).
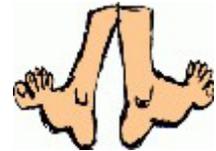
Again, we can begin by asking the user some questions to get the information of what the kids are wearing:

```
public class BarefootProgram {
    public static void main(String[] args) {
        System.out.println("How many kids are there ? ");
        System.out.println("How many are wearing socks ? ");
        System.out.println("How many are wearing shoes ? ");
        System.out.println("How many are wearing both socks AND shoes ? ");
    }
}
```

For each of these questions, we will need to ask the user for the information and then store this so that we can compute an answer afterwards.   We will need 4 integer variables to store the above numbers.   Since we will need to get more than one input, we can also create a **Scanner** object and store it in a variable so that we can re-use it many times as follows:

```java
import java.util.Scanner;

public class BarefootProgram {
    public static void main(String[] args) {
        int           numKids, withSocks, withShoes, withBoth;
        Scanner       keyboard = new Scanner(System.in);

        // Get the user input
        System.out.println("How many kids are there ? ");
        numKids = keyboard.nextInt();

        System.out.println("How many are wearing socks ? ");
        withSocks = keyboard.nextInt();

        System.out.println("How many are wearing shoes ? ");
        withShoes = keyboard.nextInt();

        System.out.println("How many are wearing both shoes and socks ? ");
        withBoth = keyboard.nextInt();
    }
}
```

Now, we need to compute and display the answer:

```java
import java.util.Scanner;

public class BarefootProgram {
    public static void main(String[] args) {
        int           numKids, withSocks, withShoes, withBoth;
        Scanner       keyboard = new Scanner(System.in);

        // Get the user input
        System.out.println("How many kids are there ? ");
        numKids = keyboard.nextInt();

        System.out.println("How many are wearing socks ? ");
        withSocks = keyboard.nextInt();

        System.out.println("How many are wearing shoes ? ");
        withShoes = keyboard.nextInt();

        System.out.println("How many are wearing both shoes and socks ? ");
        withBoth = keyboard.nextInt();

        // Now compute the answer
        int socksOnly = withSocks - withBoth;
        int shoesOnly = withShoes - withBoth;
        int barefoot = numKids - withBoth - socksOnly - shoesOnly;

        // Finally, display the results nicely
        System.out.println("There are " + barefoot + " barefoot kids.");
    }
}
```

Note that we could have simplified the above code with a different formula that does not require intermediate variables **socksOnly** and **shoesOnly** as follows:

```
barefoot = numKids -  withSocks - withShoes + withBoth;
```

In time, you will learn to make your code more concise and be able to eliminate variables that you do not need.

1.7 Calculations and Formulas

Obviously, a computer can compute solutions to mathematical expressions.   We can actually perform simple math expressions such as:

30 + 5 * 2 - 18 / 2 – 2

In such a math expression, we need to understand the order that these calculations are done in.   You may recall from high school the **BEDMAS** memory aid which tells you to perform **B**rackets first, then **E**xponents, then **D**ivision & **M**ultiplication, followed by **A**ddition and **S**ubtraction.

So, for example, in the above JAVA expression, the multiplication * operator has preference over the addition + operator. In fact, the * and / operators are evaluated first from left to right and then the + and -.   Thus, the step-by-step evaluation of the expression is:

30 + **5 * 2** - 18 / 2 - 2
30 + 10 - **18 / 2** - 2
30 + 10 - 9 - 2
40 - 9 - 2
31 - 2
29
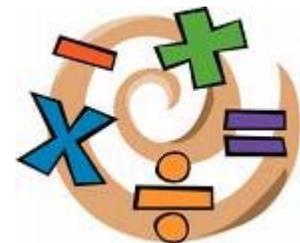
We can always add round brackets (called *parentheses*) to the expression to force a different order of evaluation. Expressions in round brackets are evaluated first (left to right):

(30 + 5) * (2 - (18 / 2 - 2))
35 * (2 - (**18 / 2** - 2))
35 * (2 - **(9 - 2)**)
35 * **(2 - 7)**
35 * -5
-175

In JAVA, it is good to add round brackets around code when it helps the person reading the program to understand what calculations/operations are done first.

Another operator that is often useful is the *modulus* operator which returns the remainder after dividing by a certain value.   In JAVA we use the **%** sign as the modulus operator:

```
10 % 2        // results in the remainder after dividing 10 by 2 which is 0
10 % 3        // results in the remainder after dividing 10 by 3 which is 1
10 % 4        // results in the remainder after dividing 10 by 4 which is 2
39 % 20       // results in the remainder after dividing 39 by 20 which is 19
```

Note that using a modulus of 2 will allow you to determine if a number is an odd number or an even number … which may be useful in some applications.

In addition to the standard math operations (i.e., **+ , - , * , /** and **%**), there are some math operators that can reduce the amount of code that you need to write.   For example, the following code outputs the values 8 and 9 to the console window:

```
int    n = 8;

System.out.println(n);

n = n + 1;

System.out.println(n);
```

There is an increment operator called **++** which is a quick way to add 1 to a variable. The following code does the same thing:

```
int    n = 8;

System.out.println(n);

n++;       // same as n = n + 1

System.out.println(n);
```

In fact, this short form of incrementing a variable by 1 is often used within other JAVA expressions.   For example, the following produces the same result:

```
int    n = 8;

System.out.println(n++);
System.out.println(n);
```

Here, the **++** is considered a *post-operator* in that it increments **n** *after* it is used in the expression (i.e., after it is printed).   The ++ can also be used as a *pre-operator* by placing it in front of the variable so that the value of the variable is incremented *before* it is used in the expression.   Hence, the following code produces the same result of 8 and 9:

```
int    n = 8;

System.out.println(n);
System.out.println(++n);
```

Similarly, there is a **--** post-operator/pre-operator that can be used to *decrement* a variable.

In addition to these operators, there are some binary assignment operators that perform an operation on some numbers and also assign the new value to the variable.   For example, consider the following code which outputs 8 and 80 to the console:

```
int    n = 8;

System.out.println(n);

n = n * 10;

System.out.println(n);
```

We can replace n = n * 10 with a shorter form which does the same thing as follows:

```
int    n = 8;

System.out.println(n);

n *= 10;          // same as n = n * 10

System.out.println(n);
```

This code multiples **n** by 10 and then puts the result back into **n** again.   There are similar binary operators for the other standard operations: **+=**, **-=**, **/=** and **%=**.

If you want to do something beyond these simple operations, you may want to look at the **Math** library in JAVA.   The following is a list of *some* of the more useful functions in the **Math** library that can be used on numbers:

Trigonometric:

- **Math.sin**(0)   // returns 0.0 which is a double
- **Math.cos**(0)   // returns 1.0
- **Math.tan**(0.5)   // returns 0.5463024898437905
- **Math.PI**   // returns 3.141592653589793
  (note that **Math.PI** has no brackets … because it is a fixed constant value, not a function)

Conversion and Rounding:

- **Math.round**(6.6)   // returns 7
- **Math.round**(6.3)   // returns 6
- **Math.ceil**(9.2)   // returns 10
- **Math.ceil**(-9.8)   // returns -9

- **Math.floor**(9.2)   // returns 9
- **Math.floor**(-9.8)   // returns -10
- **Math.abs**(-7.8)   // returns 7.8
- **Math.abs**(7.8)   // returns 7.8

Powers and Exponents:

- **Math.sqrt**(144)   // returns 12.0
- **Math.pow**(5,2)   // returns 25.0
- **Math.exp**(2)   // returns 7.38905609893065
- **Math.log**(7.38905609893065)   // returns 2.0

Comparison:

- **Math.max**(560, 289)   // returns 560
- **Math.min**(560, 289)   // returns 289

Generation of a Random Number:

- **Math.random**()   // returns a double >=0.0 and <1.0

These functions are used just as shown above.   Notice that we need to write **Math.** in front of all of these functions in order to use them.   This is the way we tell JAVA that we want to use the functions in the **Math** library.   In fact, **Math** is just one of the many useful pre-defined classes in JAVA.   We do not need to import the **Math** class because it is in the **java.lang** package and is thus automatically imported.

As an example, consider how to write a program that computes the volume of a ball (e.g., how much space a ball takes up).

How would we write a JAVA code that computes and displays the volume of such a ball with radius of **25cm** ?
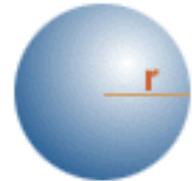
We need to understand the operations.   We need to do a division, some multiplications, raise the radius to the power of 3 and we need to know the value of **π** (i.e., pi).

$$Volume = \frac{4\pi r^3}{3}$$

Here is the simplest, most straight forward solution:

```
int r = 25;
System.out.println((4 * Math.PI * Math.pow(r, 3) / 3));
```

This would also have worked, but requires the radius **r** to be duplicated:

```
System.out.println((4 * Math.PI * (r*r*r) / 3));
```

We could even substitute our own value for **π** :

```
System.out.println((4 * 3.141592653589793 * (r*r*r) / 3));
```

Alternatively, we could have evaluated the 4/3 first:

```
System.out.println((4/3 * Math.PI * Math.pow(r, 3)));
```

Or even pre-compute 4 π /3 (which is roughly 4.188790204786) :

```
System.out.println((4.188790204786 * Math.pow(r, 3)));
```

The point is that there are often many ways to write out an expression.   You will find in this course that there are many solutions to a problem and that everyone in the class will have their own unique solution to a problem (although much of the code will be similar because we will all usually follow the same guidelines when writing our programs).

At the end of this chapter, there is a table that shows of a few other mathematical operators that you may wish to use in the future.   Also, it shows the order that JAVA uses to evaluate its operators.   You are not responsible for knowing or memorizing anything in that table ... it is just for your own personal use.

1.8 Type Conversion

When programming, we often find ourselves working with different kinds of data.   For example, even when performing simple calculations, we may end up using a variety of primitive data types.

```
float    price;
int      payment;
double   taxes, change;

price = 34.56f;
taxes = price * 0.13;
payment = 50;

change = payment - price - taxes;
```

Notice that the above code performs calculations using **ints**, **floats** and **doubles**.   When performing such calculations, JAVA performs some automatic *type-conversion*.   That is, it converts one type of data into another when performing the calculation.

During computations, JAVA will always produce its calculated result as being the same type as the *more precise* data type that was used in the calculation.   In the above example, doubles are more precise than **floats** and **ints**.   Therefore, when we do **price * 0.13**, JAVA notices that this is a calculation using a **float** (less precise) and a **double** (more precise).   Therefore the **float** is converted to a **double** during the computation and the resulting answer is returned as a **double**, and stored in the **taxes** variable.

Consider the difference in output of the following code:

```
float       price1 = 34.56f;
double      price2 = 34.56;
System.out.println(price1 * 0.13f);   // displays 4.4928
System.out.println(price2 * 0.13f);   // displays 4.492799835205078
```

Notice that the same calculation is performed in both cases but that one uses a **float** price amount while the other uses a **double** price amount.   The 1st calculation uses two **floats**, and so the result is a less precise **float** value.   The 2nd calculation uses a **double**, so the entire calculation is performed using **doubles**, generating a more precise result.

What if we changed the code to store the results as follows:

```
float       price1 = 34.56f;
double      price2 = 34.56;
float       result;

result = price1 * 0.13f;
result = price2 * 0.13f;     // gives "possible loss of precision" error
```

The above code will not compile.   JAVA notices that in the last line, it is performing a calculation that will result in a **double**.   However, **result** is of type **float**.   Since **floats** are less precise that **doubles**, the JAVA compiler informs us that there would be a loss of precision if we tried to take the **double** answer and "squeeze" it into a smaller **float** variable.   Basically, we cannot squeeze a big thing into a small box.

When assigning calculation results to a variable, JAVA always checks to make sure that the resulting type of the calculation will "fit" into the variable.   We **CANNOT** store a …

- **double** result in a variable of type **float**, **int**, **long**, **byte**, **short**
- **float** result in a variable of type **int**, **long**, **byte**, **short**
- **long** result in a variable of type **int**, **byte**, **short**
- **int** result in a variable of type **byte**, **short**
- **short** result in a variable of type **byte**

If we attempt to assign a result into a variable of a less precise type (as above), then we will ALWAYS get a compiler error stating "possible loss of precision".

However, we **CAN** store a:

- **double** result in a variable of type **double**
- **float** result in a variable of type **float, double**
- **long** result in a variable of type **long, float, double**
- **int** result in a variable of type **int, long, float, double**
- **short** result in a variable of type **short, int, long, float, double**
- **byte** result in a variable of type **byte, short, int, long, float, double**

Sometimes, however, we may want to take a more precise calculated value and store it into a less precise variable, perhaps for later use.   For example, we may want to perform a money-based calculation precisely but then we may only be interested in the whole number portion, or maybe only 2 decimal places.   This example calculates the change owed to a person, extracts and stores the whole portion (as whole monetary bills to be returned to the customer … ignore the fact that toonies and loonies are not bills) and the remaining change as a separate value:

```
float    price, changeDue;
int      payment, billsDue;
double   taxes, change;

price = 34.56f;
taxes = price * 0.13;
payment = 50;

change = payment - price - taxes;
billsDue = (int)change;
changeDue = (float)change - billsDue;
System.out.println(change);        // displays 10.947198448181151
System.out.println(billsDue);      // displays 10
System.out.println(changeDue);     // displays 0.94719887
```

Notice that we used **(int)** and **(float)**.   These are called ***explicit type-casts***. In English, the term ***typecasting*** means *to identify as belonging to a certain type*.   What we are doing is telling the JAVA compiler that we would like to "convert" a particular value into the type that we specified between parentheses **( )**.

We can typecast any numeric type (i.e., **double, float**, **int**, **long**, **byte**, **short**, **char**) to any other numeric type at any time. We just need to remember what happens each time as follows:

- if **x** is a **double** or **float** then **(long)x, (int)x, (short)x, (byte)x** and **(char)x** will discard the decimal places … it will NOT round off, just truncate.

- if **x** is a **long**, **int**, **short**, **byte** or **char**, then **(double)x** and **(float)x** will set the decimal places to be **.0**.

- if a more precise value (e.g., **long** or **double**) is type-casted to a less precise value (e.g., **int** or **double**) then some data WILL be lost.

Here are some examples in which the conversion results in data loss:

```
(float)34.56767867    ==>    34.56768  // rounded off
(int)2.4              ==>    2         // decimal places lost
(int)2.9              ==>    2         // does not round off
```

Here are some examples in which the conversion results in data loss:

```
(char)947384                  ==>    '?'
(int)123456789012345678L      ==>    -1506741426
```

Conversions may be intermediate and unintentional:

```
int       sum = 30;
double    avg = sum / 4;  // result is 7.0, not 7.5 !!!
```

Perhaps a more common type of conversion is that of converting numbers to Strings and vice-versa.   In JAVA, there are some pre-defined functions to do this for us.

In order to convert a given **String** to a particular numeric data type, there are various functions available:

```
String    s = ...;

Integer.parseInt(s);  // returns int value of s
Double.parseDouble(s) // returns double value of s
Float.parseFloat(s)   // returns float value of s
```

Here are some examples:

```
Integer.parseInt("7438");         // returns int value 7438
Double.parseDouble("234.65")      // returns double value 234.65
Float.parseFloat("234.65")        // returns float value 234.65f
```

We sometimes need to use these functions if we are given a **String** from the user and would like to convert the input string to a numeric value for calculation purposes.   For example, the following code asks the user for his/her age and then uses the input to determine the number of years until their retirement:

```
String     input;
int        age;

input = JOptionPane.showInputDialog("What is your age ?");
age = Integer.parseInt(input);

JOptionPane.showMessageDialog(null, "You have " + (65 – age) +
                             " years until retirement");
```

Finally, we would also like to be able to convert in the other direction.   That is, perhaps we would like to convert a number to a String.   This is often necessary in order to place numeric information into a text field on a window.   The simplest way to do this is to take the **int/float/double/long** value and simply add it to an empty String.   JAVA will then convert it:

```
int       age;
String    s;

age = 21;
s = age;          // compile error:   incompatible types
s = "" + age;     // this will work
```

Another option is to use any of the following functions:

```
Integer.toString(225)         ==> "225"
Double.toString(225.56)       ==> "225.56"
Float.toString(225.56f)       ==> "225.56"

Integer.toBinaryString(225)  ==> "11100001"
Integer.toHexString(34728)   ==> "87a8"
Integer.toOctalString(34728) ==> "103650"
```

Notice that the last 3 are quite useful because they actually change the appearance of the integer value within the string according to the desired number system.

1.9 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:

```java
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).

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:

```java
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:                  Enter product price:
 35.99                                 8.85
 Total cost:$40.67                     Total cost:$10.00
 Enter payment amount:                 Enter payment amount:
 50                                    10
 Change:$9.33                          Change:$-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 …

```java
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:

```java
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**).

At the end of this chapter, there is a table that shows 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.

Supplemental Information (Mathematical Operators)

JAVA also provides *bitwise operators* for integers and booleans:

- **~** bitwise complement (prefix unary operator)
- **&** bitwise and
- **|** bitwise or
- **^** bitwise exclusive-or
- **<<** shift bits left, filling in with zeros
- **>>** shift bits right, filling in with sign bit
- **>>>** shift bits right, filling in with zeros

To understand how these work, you must understand how the numbers are stored as bits in the computer. We will not discuss bit manipulation in this course.

Here is a table showing the operators in JAVA (some which we have not yet discussed) and their *precedence* (i.e., the order that they get evaluated in). The topmost elements of the table have higher precedence and are therefore evaluated first (in a left to right fashion). In the table, **<exp>** represents any JAVA expression. If however, you are writing code that depends highly on this table, then it is likely that your code is too complex.

postfix operators	**[] . () ++ --**
prefix operators	**++ -- - ~ !**
creation/cast	**new (<typecast>)**
multiplication/division/modulus	*** / %**
addition/subtraction	**+ -**
shift	**<< >> >>>**
comparison	**< <= > >= instanceof**
equality	**= = ! =**
bitwise-and	**&**
bitwise-xor	**^**
bitwise-or	**|**
logical and	**&&**
logical or	**||**
conditional	**<bool_exp>? <true_val>: <false_val>**
assignment	**=**
operation assignment	**+= -= *= /= %=**
bitwise assignment	**>>= <<= >>>=**
boolean assignment	**&= ^= |=**

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.