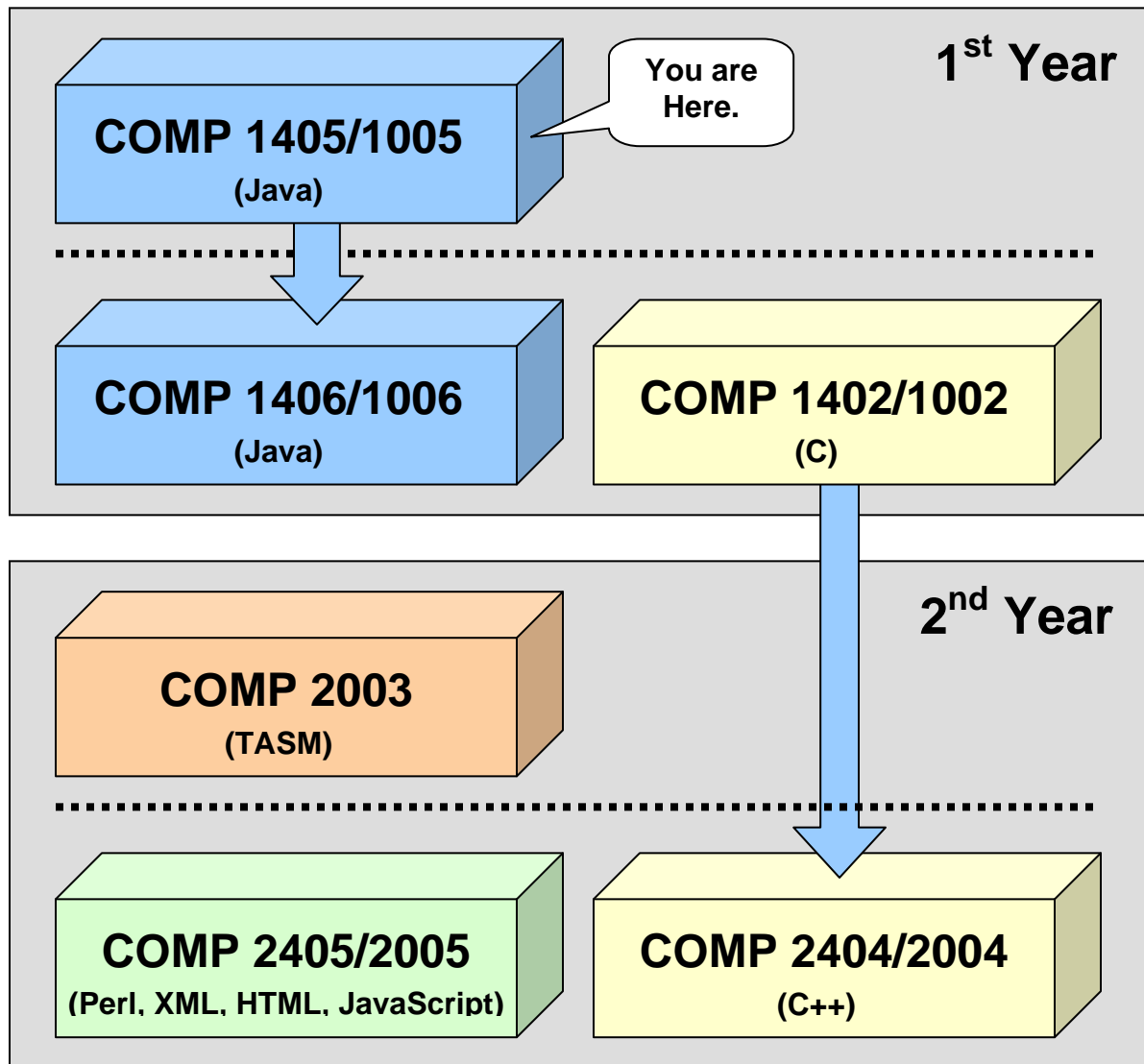# Programming Basics

What is in This Chapter ?

This first chapter explains very briefly the difference between Traditional vs. Object-Oriented programming and how the **JAVA** programming language basically works. It then discusses some of the basic concepts behind Object-Oriented programming and how to write a simple "Hello World" program in JAVA. 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. Lastly, since JAVA is NOT purely Object-Oriented (i.e., it contains basic data types called *primitives* which form the basis for all data stored in objects), we will also look at these primitives as basic building blocks in JAVA and how they are used to form Objects.

1.0 Programming-Related Courses In Computer Science

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 programming courses:

1st Year

COMP 1405/1005
(Java)

You are Here.

COMP 1406/1006
(Java)

COMP 1402/1002
(C)

2nd Year

COMP 2003
(TASM)

COMP 2405/2005
(Perl, XML, HTML, JavaScript)

COMP 2404/2004
(C++)

Of course, there are other computer science courses as well. These are just the core courses that nearly everyone is required to take.   After this course is over, you *should* understand how to write computer programs.   You will also understand what it means to do ***object-oriented programming***.   In the winter term, you will take COMP1406/1006 which is like a continuation of this course.   Together, these two courses 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.

1.1 Understanding Programming

What are Computers Used For ?

- Communications: Internet, e-mail, cell phones
- Word Processing: typing/printing documents
- Business Applications: accounting, spreadsheets
- Engineering Applications: scientific analysis, simulations
- Database Management: police records, stock market
- Entertainment: games, multimedia applications
- Manufacturing: CAD/CAM, robotics, assembly
- ... many more ...

Who is Involved With Computers ?

- *System/Hardware Designers* = people that design computers and related products.
- *Manufacturers* = people that actually build and assemble computers.
- *Software Designers* = people that design applications to be used with the computers.
- *Programmers* = people that write computer programs to achieve working applications, games and other software packages.
- *End User* = people that buy and use the software when it is done.

We are going to play the role of the *Programmer* in this course. 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 user is happy with software that is not full of bugs (i.e., problems)!

What is a *program* ?

A *program* is traditionally known as:

a sequence of instructions that can be executed by a computer to solve some problem.

In this course, we will learn to write our own programs to solve some very simple problems. Writing programs is often called "writing code". So the *code* is actually the program itself. We also use the term "*source code*" to represent the program logic.
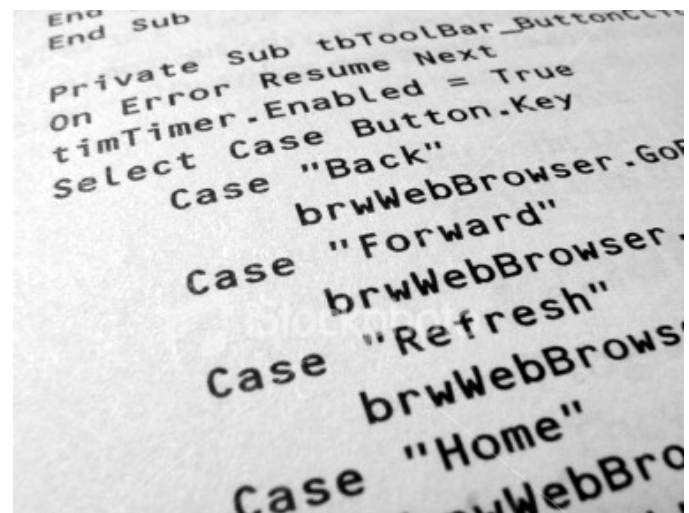
How do we write *good* programs ?

There are some important standards which you should always adhere to if you want to write "good" programs:

- **C** orrectness - Make sure that your program does what it is supposed to do.
- **R** obustness - Make sure the program does not crash. Testing helps prevent this.
- **I** nterface Usability - Make sure that the interface is easy to use and intuitive.
- **S** implicity - Keep the code as simple as possible, while still meeting requirements.
- **P** resentation and **Documentation** - Make sure that software is thoroughly documented for maintenance purposes. (Don't forget about the TA's that need to mark your assignments)
- **E** fficiency - Make sure that the software runs fast and does not use up too much computer memory (i.e., that it is *time* and/or *space* efficient).

There are other issues that may need to be considered when writing programs:

- *Portability* – The ability to run your program on different kinds of machines (e.g., Windows PC, Mac, Sun Workstation, etc…)
- *Security* – The need to make sure that information entered by the user is not visible to anyone who wants it (e.g., VISA or bank account info, passwords, personal data, etc…)

What is a *programming language* ?

To write a program, we need to use what is called a *programming language* which is:

- *an artificial language designed to automate the task of organizing and manipulating information, and to express algorithms (i.e., problem solutions) precisely.*

A programming language "boils down to" a set of words, rules and tools that are used to explain (or define) your program.   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#, Smalltalk)

There are also other types of programming languages such as **functional** programming languages and **logic** programming languages. According to WikiPedia, as of 2008 the 12 most actively used programming languages are (in alphabetical order): **C**, **C++**, **C#**, **Java**, **JavaScript**, **Perl**, **PHP**, **Python**, **Ruby**, **Shell**, **SQL** and **VisualBasic**.

What is *Procedural* programming ?

Procedural Programming involves writing code line-by-line in the order that the code must be executed (i.e., run or evaluated) to solve some problem. That is, the order of the set of steps is determined beforehand in a way that solves the problem logically.   All programming languages involve writing "some" code in a procedural manner.   This is because, to solve most problems, there is usually a set of steps that must be followed in order.   In the real world, we also follow steps to accomplish a task. For example, to withdraw some money from a bank machine we usually follow these steps:
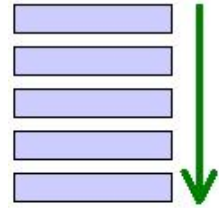
1. insert your bank card
2. enter your PIN #
3. select "withdraw"
4. enter amount to withdraw … press "ok"
5. take your money and your card

Usually, the problem that is trying to be solved by the computer is broken down into smaller ***sub-problems*** and these are solved in turn.   For such simple structured programs, they often follow these steps:

1. Get input data from the user (e.g., name, account#, deposit amount, etc.)
2. Perform some calculations, make some decisions, store/change some data (e.g., adjust account balance)
3. Display some data output on the screen (e.g., tell user what happened)

What is *Object-Oriented* programming ?

Object-Oriented Programming (a.k.a., OOP) is similar to that of procedural programming in that it involves executing a set of instructions in some specified order.   However, it differs from procedural programming in the way that your code is organized.   Programming using object-oriented *style,* involves organizing your code in "chunks" that logically correspond to real-world objects.   For example, you may group all of your code related to a *person* into one file (called a **_class_**) while code related to a *car* or a *bank account* would be grouped together in separate files (i.e., classes).

When doing OOP, the programmer (i.e., you) spends much time ***defining*** (i.e., writing code for) various objects by specifying small/simple behaviors that the object will need to respond to (e.g., deposit, withdraw, compute interest, get age, save data etc...)   Hence, OOP is all about knowing:

- which objects to use,
- the kind of information we need to know about the objects,
- how objects behave, and
- how to use them with each other

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
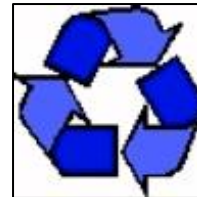
Encapsulation

- 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 the JAVA language.

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)
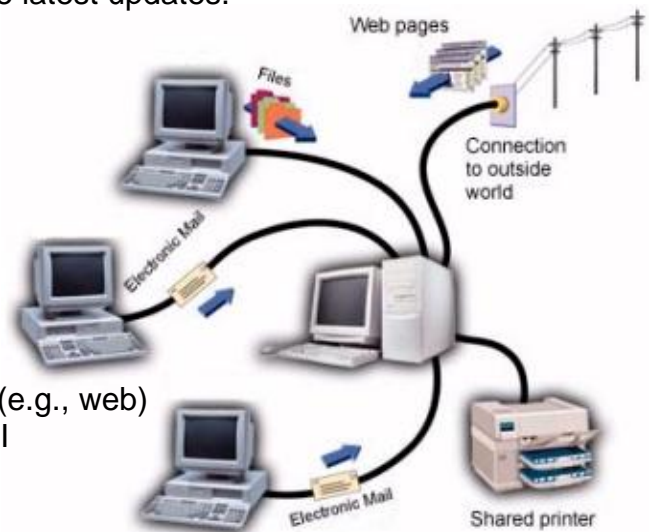
1.2 The JAVA Programming Language

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.java.sun.com website for the latest updates.

There are many reasons to use JAVA:

- **architecture independence**
    - o ideal for internet applications
    - o code written once, runs anywhere
    - o reduces cost $$$

- **distributed and multi-threaded**
    - o useful for internet applications
    - o programs can communicate over network (e.g., web)
    - o uses RMI (Remote Method Invocation) API

- **dynamic**
    - o code loaded only when needed

- **memory managed**
    - o automatic memory allocation / de-allocation
    - o garbage collector releases memory for unused objects
    - o simpler code & less debugging

- **robust**
    - o strongly typed
    - o automatic bounds checking
    - o no "pointers" (you will understand this in COMP1402/1002)

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) from Sun Microsystems which you can download from the SUN website 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 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...)

We will be using the **JCreatorLE** IDE within this course (from www.jcreator.com).   It is a powerful, yet simple "Windows-based" IDE that fits all of our needs.   As you become a more skilled programmer, you may wish to switch over to the **Eclipse** IDE (from www.eclipse.org) which has additional features... but is overly complex for beginning students. If you are using a **Mac** computer, **jGRASP** (from www.jGRASP.org) is a nice simple IDE as well.

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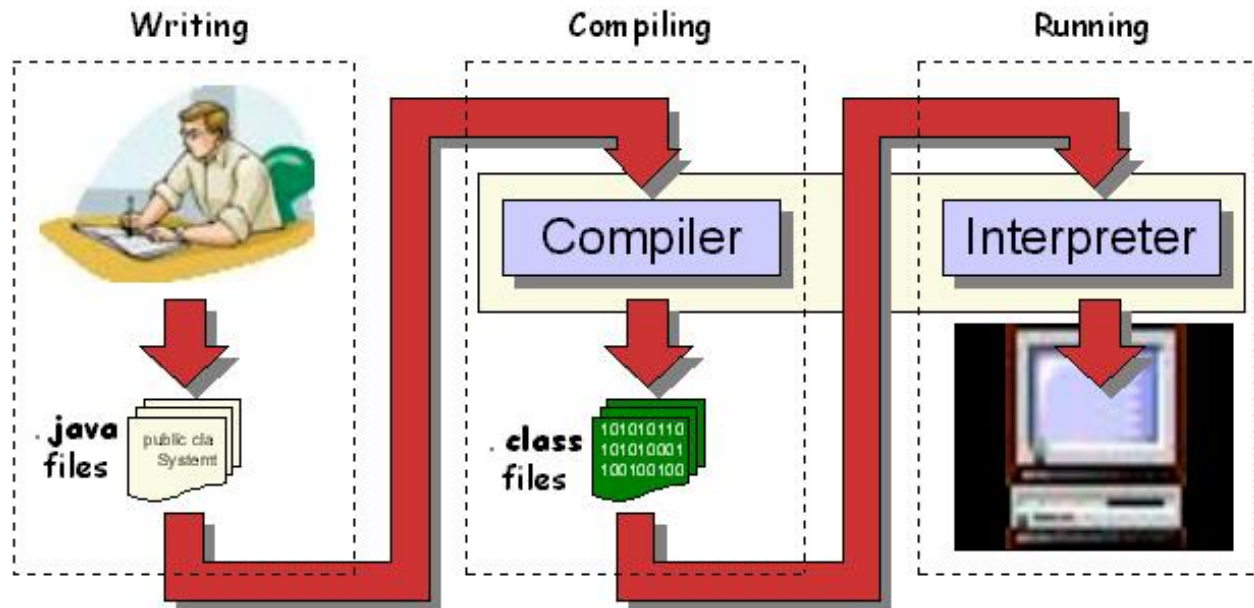
The java *compiler*:
- prepares your program for running
- produces a **.class** file containing ***byte-codes*** (which is a program that is ready to run).

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:

```
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).

- The first line beings with word **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:

```
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.

Where do we write this code ?   In the JCreator IDE.   JCreator has a main window area into which we write the code and another area at the bottom where we view the results:
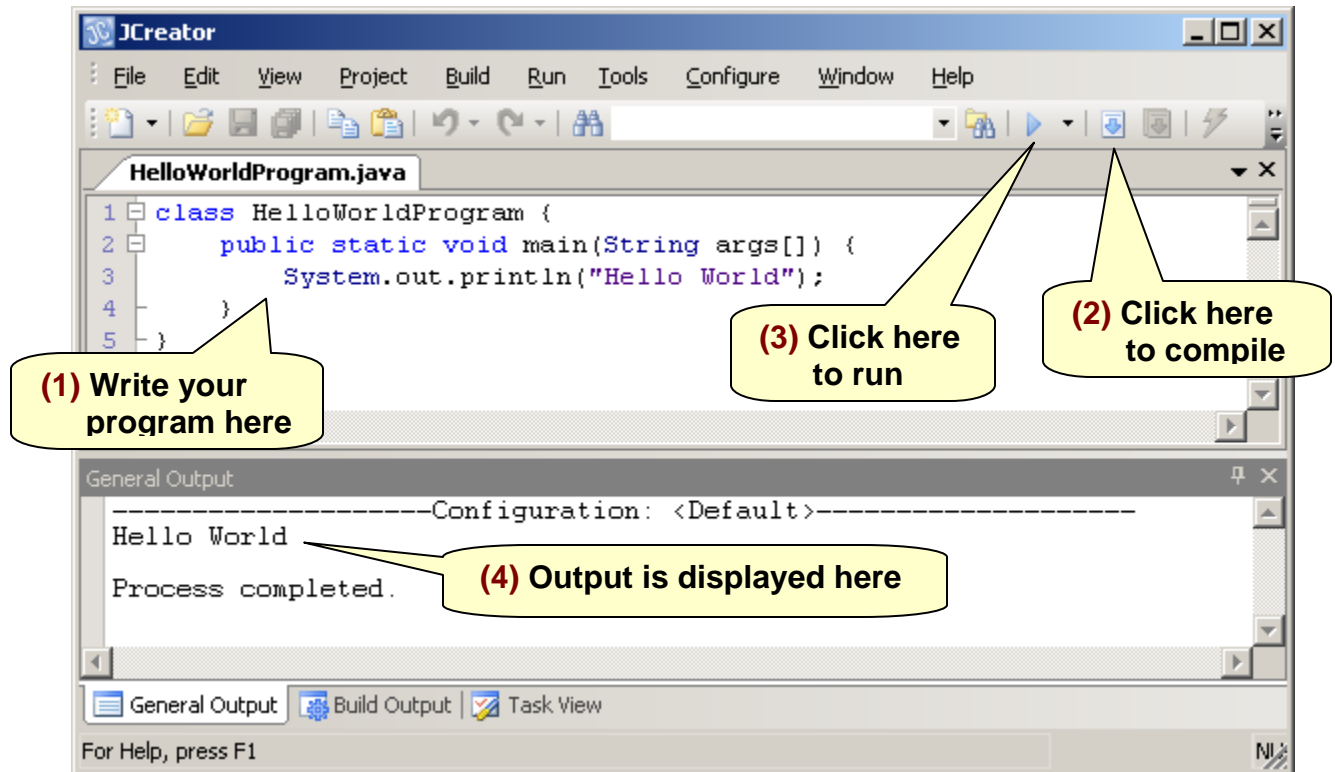
Supplemental Information (Capturing Screen Output)

If by default, your JCreator window opens up a black **Command Prompt** window when compiling or running, you will need to configure JCreator to allow the compiled results and the program output to appear at the bottom of your JCreator window.   Do the following steps (you only need to do this once):

1.   Click on the **Configure** menu on the menu bar.
2.   Select **Options...** and a dialog box will appear.
3.   Select **JDK Tools** from the left list (2/3 of the way down the list).
4.   Make sure that the **Select Tool Type:** is set to **Compiler** in the drop down list.
5.   Select **<Default>** in the list below (even though it appears as grayed out).
6.   Press the **Edit...** button.
7.   Click "on" the **Capture output** checkbox (i.e., a check mark will appear).
8.   Press OK.
9.   Now choose from the **Select Tool Type:** drop down list the option of **Run Application**.
10.   Select **<Default>** in the list below (even though it appears as grayed out).
11.   Press the **Edit...** button.
12.   Click "on" the **Capture output** checkbox (i.e., a check mark will appear).
13.   Press OK.
14.   Press OK again to close the dialog box.

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
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.

    o Generally, you should use **//** when you have a single line to comment (i.e., everything after the **//** characters on that line is ignored).
    o 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 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 JCreator 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:
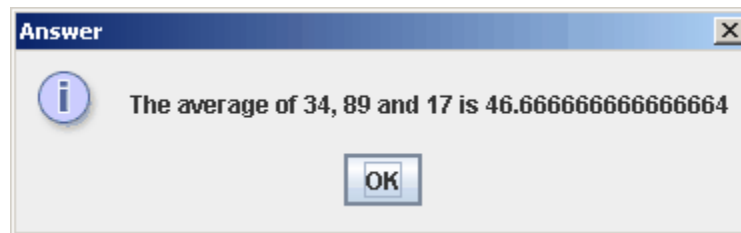
The window would come up, wait for us to press the **OK** button and then close.   Here is the code that does this:

```java
import javax.swing.JOptionPane;

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;

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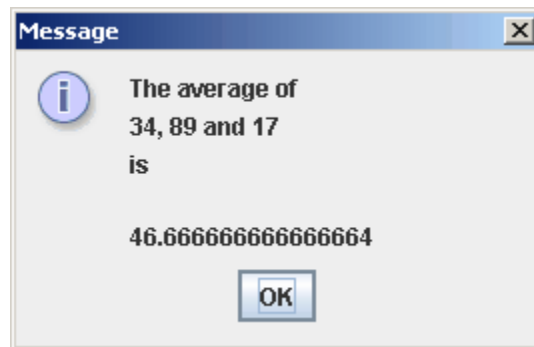
Of course, the appearance is not as pleasant because everything is aligned to the left.   As is, there is no way to change this alignment. Nevertheless, it is sometimes nice to be able to display text in our own windows as opposed to simply in the console window.

In the follow-up course to this one (COMP1406/1006), 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.
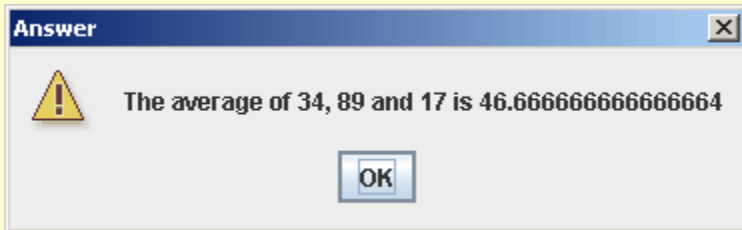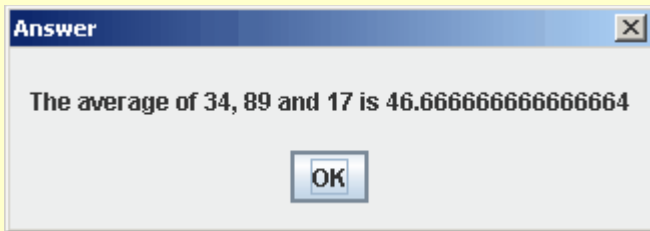
JOptionPane.WARNING_MESSAGE
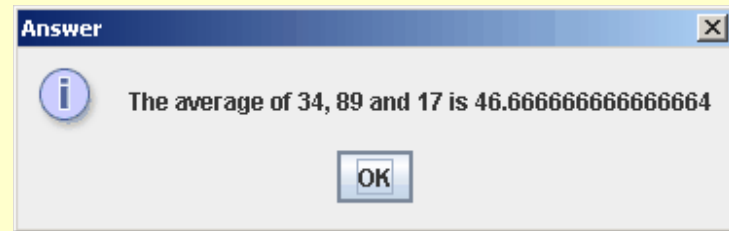Use this when you want to warn the user about something in the program.
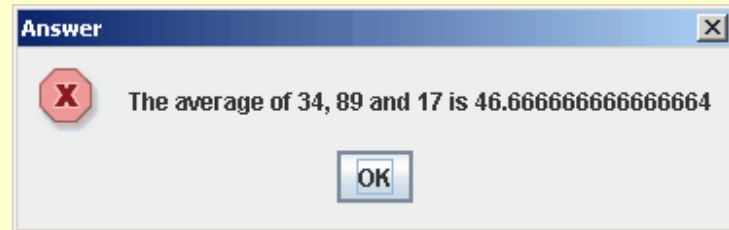
JOptionPane.ERROR_MESSAGE
Use this when you want to tell the user that an error has occurred in the program.

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.   For many years, previous versions of JAVA did not have any nice clean way to read data from the keyboard.   Since version 1.5 however, a new class was created which allows simplified input from the keyboard.   The class is called **Scanner** and it is available in 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 name and then says hello to him/her:

```
import java.util.Scanner;

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;

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:

```
import javax.swing.JOptionPane;


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 now:

```
   Hello, Mark
```

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

```java
import javax.swing.JOptionPane;

class WindowGreetingProgram2 {
    public static void main(String args[]) {
        JOptionPane.showMessageDialog(null, "Hello, " +
                        JOptionPane.showInputDialog("What is your name ?"));
    }
}
```
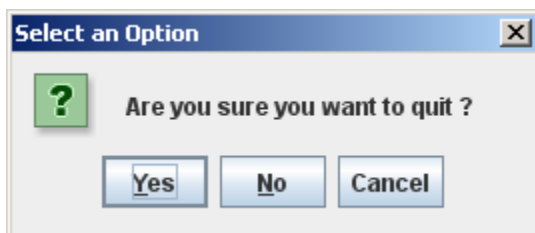
Then we end up with windows for input and for output … with nothing printed to the console:

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 Types of Data: Primitives vs. Objects

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.

For example, recall our example for averaging 3 numbers entered by the user.   What if we then wanted to find the <u>maximum</u>, the <u>minimum</u> and perhaps the sales <u>tax</u> 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.

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. 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.   Whenever we want to store any data whatsoever, we will always need to specify the *type* of that data.

So, when programming, we will define all of our data as these higher level things which we call *data types*. All data in JAVA is stored as one of the following two data types:

1. *Primitive*:   represents a **single** piece of information
                  such as a number or character.

2. *Object*:   represents **multiple** pieces of information
              that are grouped together in a way similar to placing an
              elastic around a bunch of primitives to hold them together.
              So put simply, objects are *bundles of data.* As you will see
              later, an object can actually be made up of other objects as
              well.

In JAVA there are exactly **8** primitive types.   Each differs with respects to:

- the *kind of information* being stored
- the *amount of space* they take up in the computer's memory.

You should use the data type that suits your needs but does not waste memory by using more space than is necessary.

If you want to store a simple **integer** number (i.e., no decimal part), JAVA offers you 4 possibilities:

Type	Space Used (bytes)	Stores Integer Number in 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

If you want to store a **real** number (i.e., with a decimal part), JAVA offers you 2 possibilities:

Type	Space Used (bytes)	Stores Real Number in Range
float	4	a real number over 10^{38}
double	8	a real number over 10^{308}

If you want to store a single **character** you have one choice:

Type	Space Used (bytes)	Stores Single Character
char	2	Represented by single quotes (e.g., 'a' 'B' 'c' '$' '>')

Finally, if you want to store a single **true/false** value, you also have one choice:

Type	Space Used (bytes)	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.

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**.

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

```java
class PrimitiveTestProgram {
    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
    }
}
```

Here is the resulting output:

```
239
823100267876
3.141592653589793
3.1415927
A
5

"


true
false
```

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.

Now that we have taken a look at the most primitive forms of data, lets see how we can use objects in our programs.

Objects

We have already seen how to make **Strings** of characters and how to display them to the screen.   **Strings** are *objects*.   So we have already begun using objects in our programs. What other kinds of objects are there ?   Well, JAVA has a lot of pre-defined objects.

```java
class ObjectTestProgram {
    public static void main(String args[]) {
        System.out.println(new java.lang.Object());      // general object
        System.out.println(new java.lang.String());      // blank string
        System.out.println(new java.util.Date());        // date object
        System.out.println(new java.awt.Point(50, 75)); // point object
        System.out.println(new java.awt.Rectangle(5,10,20,30)); // rectangle
    }
}
```

You may notice that for each object, we use the word **new**.   This is required because objects have to be *constructed* (i.e., built or created) before we can use them.   In this case, we are just creating the objects and then displaying them.   Notice that the **Point** and **Rectangle** objects required additional information (called *parameters*) to define them when they were created.   That is, it does not make sense to define a point without both **x** and **y** values. And a rectangle cannot be created unless we know the top left corner (e.g., x=5,y=10) and the width and height (e.g., Width=20, Height=30).

Notice how these objects are displayed in the output:

```
java.lang.Object@3e25a5

Mon Apr 06 11:40:51 EDT 2009
java.awt.Point[x=50,y=75]
java.awt.Rectangle[x=5,y=10,width=20,height=30]
```

Each object displays itself differently.   Notice that the **Date** object that was created actually

corresponds to today's date.   Also, notice that the **String** object was actually an empty string (i.e., no characters were displayed).

Please as well notice that after the word **new**, all of the objects have either **java.lang.**, **java.util.** or **java.awt.** specified.   That is because all of JAVA's objects are logically organized into subfolders (called *packages*).   As it turns out, **Object** and **String** objects are in the default folder called **java.lang**, **Date** objects are in the **java.util** folder and **Point** & **Rectangle** objects are in the **java.awt** folder.   If we do not specify where to find the objects, JAVA can become confused when compiling our code and will generate compile errors.

Having to specify the folders each time we use an object is a little cumbersome and tedious. We can actually simplify the code a little by using the **import** statement at the top of our program.   The **import** statement allows us to specify where to find each of the objects that we are planning to use. In that way, the code becomes simpler.   Notice the difference:

```
import java.lang.Object;
import java.lang.String;
import java.util.Date;
import java.awt.Point;
import java.awt.Rectangle;

class ObjectTestProgram2 {
    public static void main(String args[]) {
        System.out.println(new Object());                // general object
        System.out.println(new String());                // blank string
        System.out.println(new Date());                  // date object
        System.out.println(new Point(50, 75));           // point object
        System.out.println(new Rectangle(5,10,20,30));   // rectangle object
    }
}
```

Notice now that the code is less cluttered.   In fact, all classes in the **java.lang** package are automatically imported so we do not need the first two import statements.   Also, when we have multiple classes being imported from the same package (e.g., **Point**, **Rectangle**), we can use a single **import** statement with the ***** wildcard character to tell JAVA to import any needed classes from that package.   (Note that the import statement does not *load* any classes, it just instructs the compiler *where to find* them).   So here is the simplest form of the code:

```
import java.util.*;
import java.awt.*;

class ObjectTestProgram3 {
    public static void main(String args[]) {
        System.out.println(new Object());                // general object
        System.out.println(new String());                // blank string
        System.out.println(new Date());                  // date object
        System.out.println(new Point(50, 75));           // point object
        System.out.println(new Rectangle(5,10,20,30));   // rectangle object
    }
}
```

1.7 Making Your Own Objects

Now that we have looked at how to use some of JAVA's pre-defined objects … lets see how you can make your own.   Lets try to make a **Car** object and a **Person** object as follows:

```
class MyObjectTestProgram {
    public static void main(String args[]) {
        System.out.println(new Car());       // car object
        System.out.println(new Person());    // person object
    }
}
```

If you were to compile the above code in JCreator, it would give you the following errors:

```
  cannot find symbol class Car
  cannot find symbol class Person
```

JAVA cannot find any libraries that define the **Car** and **Person** objects. That is because the JAVA language does not have **Car** or **Person** classes defined anywhere.   If we want to use such objects, we must define them on our own.

How do we *define* our an object of our own ?

- Each object that we define must be defined as its own **separate class**
- Each of these classes must be saved as their own **.java** file

For example, we would define a **Car** object by making a **Car.java** file that defines the class as follows:

```
class Car {

}
```

That's it!   You just need to create this file in JCreator as a new java file, then save it and compile it.   You must make sure that the file name is exactly the same (i.e., upper/lowercase letters) as the name of the class … but with a **.java** file extension.

This file represents a *class definition* of the **Car** object (i.e., it specifies (or defines) exactly what a **Car** object actually is).   If we want to define a **Person** object, we can follow the same process by making a **Person.java** file and writing/saving/compiling the following code:

```
class Person {

}
```

After doing this, our **MyObjectTestProgram** should then compile ok.

```
class MyObjectTestProgram {
    public static void main(String args[]) {
        System.out.println(new Car());
        System.out.println(new Person());
    }
}
```

Notice now the output from the program:

```
Car@19821f
Person@42e816
```

This is what objects look like by default.   They show the name of the class, then an @ symbol, and finally a strange combination of numbers and letters.

This number/letter combination represents the location (or *address*) of the object in the computer's memory. We call this the *reference*, because this memory address "*refers to*" the object. The actual value of the address is unimportant to us, however, it is important for you to understand that each time we make an object, it "uses up" a portion of the computer's memory.

Later we will see how to change the appearance of our objects so that they show more meaningful information when displayed.
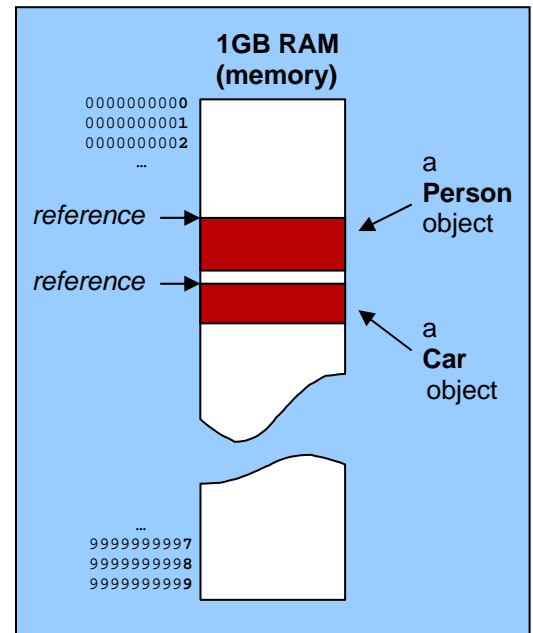
We can actually make a bunch of new **Car** and **Person** objects. Here is an example:

```
class MyObjectTestProgram2 {
    public static void main(String args[]) {
        System.out.println(new Car());          // car object
        System.out.println(new Car());          // a different car object
        System.out.println(new Person());       // a person object
        System.out.println(new Person());       // another person object
        System.out.println(new Person());       // yet another person object
    }
}
```

Notice in the output that each **Car** and **Person** has its own unique location in the computer's memory (shown by the uniqueness of the numbers/letters after the **@**):

```
Car@19821f
Car@addbf1
Person@9304b1
Person@190d11
Person@a90653
```

When we save and compile the file **Car.java** and **Person.java**, we are actually defining two new *types* of objects. We are really *defining* new *classes* (or *categories)* of objects that JAVA now understands, although no objects are actually created in memory.

Keep in mind that the **Car**, **Person** and **MyObjectTestProgram** classes are each defined in their own files and should be keep in the same folder on your hard drive.

```
class Car {

}
    class Person {

    }
        class MyObjectTestProgram {
            public static void main(String args[])
        {
                ...
            }
        `
```
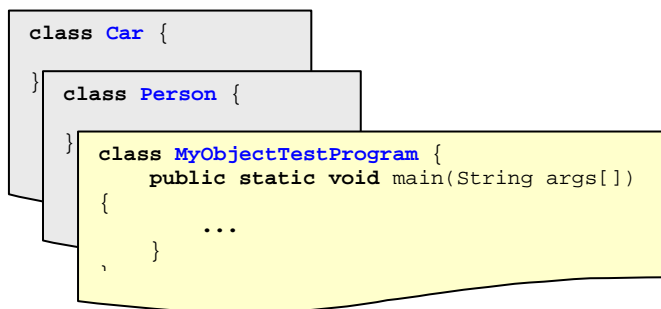
Also, remember that you cannot *run* the **Car** or **Person** class, because they are not programs … they are just object definitions.   You can only run classes that have the **public static void main(…)** code in them.

Only when we write **new Car()** … are we actually making a new object of that class type.   This is called *making an instance* of the **Car** class.   Hence, every time we write **new Car()** we are getting back an *instance* of the **Car** class.   So an **instance** is an **object**.

You may want to think of the **Car** class as a *factory* that makes **Car** objects (i.e., makes *instances* of the **Car** class). In general, every time we use **new**, JAVA goes to the factory for that class and makes an instance of that class for us and then gives it back to us. So… the *class* is the "factory", and the *instance* is the particular "object" that we can start using now in our programs.

The **Car** Class	**new Car**()	Instance of type **Car**

The **Person** Class	**new Person**()	Instance of type **Person**

The **House** Class	**new House**()	Instance of type **House**

At this point you need to understand that all data (i.e., information) in the program is represented as either one of the 8 primitives or as an object.

You must also understand that objects are just "chunks of data" of a certain type, category or class. You can use objects in your program just as easily as using simple numbers. In the next section we will find out how to start using the data (or information) in more meaningful ways.