
Chapter 1

Programming in Java

What is in This Chapter ?

This first chapter introduces you to programming JAVA applications. It assumes that you are familiar with the basics of the JAVA language syntax, which you should have learned from the previous course. In this chapter, we discuss how to make JAVA applications and the various differences between JAVA applications and the Processing applications that you may be used to writing.



1.1 The JAVA Programming Language

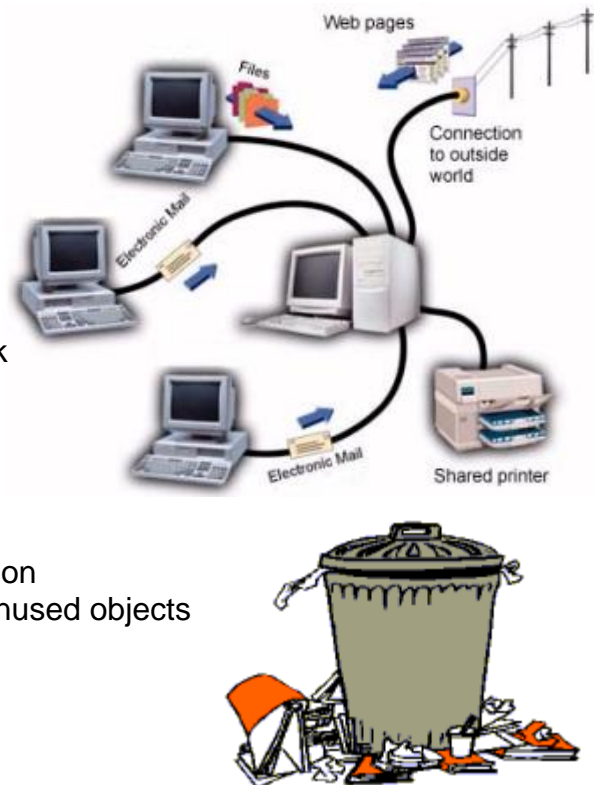
JAVA is a very popular object-oriented programming language from SUN Microsystems. In the previous course we used a language called **Processing** which is written "on top of" JAVA. That is, **Processing** uses the same syntax as JAVA. However **Processing** simplifies the process required to get a program "up and running" in that some of the overhead code is hidden from the programmer. In addition, some of the functionality in **Processing** has been simplified such as graphics and event handling ... which are just a little more complicated in JAVA. In this course, you will learn to program in JAVA and we will no longer do **Processing**.

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

- Java Media Framework (for video streaming, webcams, MP3 files, etc)
- Java 3D (for 3D graphics)
- 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/technetwork/java/index.html website for the latest updates. There are many reasons to use JAVA:

- **architecture independence**
 - ideal for internet applications
 - code written once, runs anywhere
 - reduces cost \$\$\$
- **distributed and multi-threaded**
 - useful for internet applications
 - programs can communicate over network
- **dynamic**
 - code loaded only when needed
- **memory managed**
 - automatic memory allocation / de-allocation
 - garbage collector releases memory for unused objects
 - simpler code & less debugging
- **robust**
 - strongly typed
 - automatic bounds checking
 - no "pointers" (you will understand this in when you do **C** language programming)



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



When programming in JAVA, you will usually use:

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

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

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

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

How do you get started in JAVA?

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

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

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

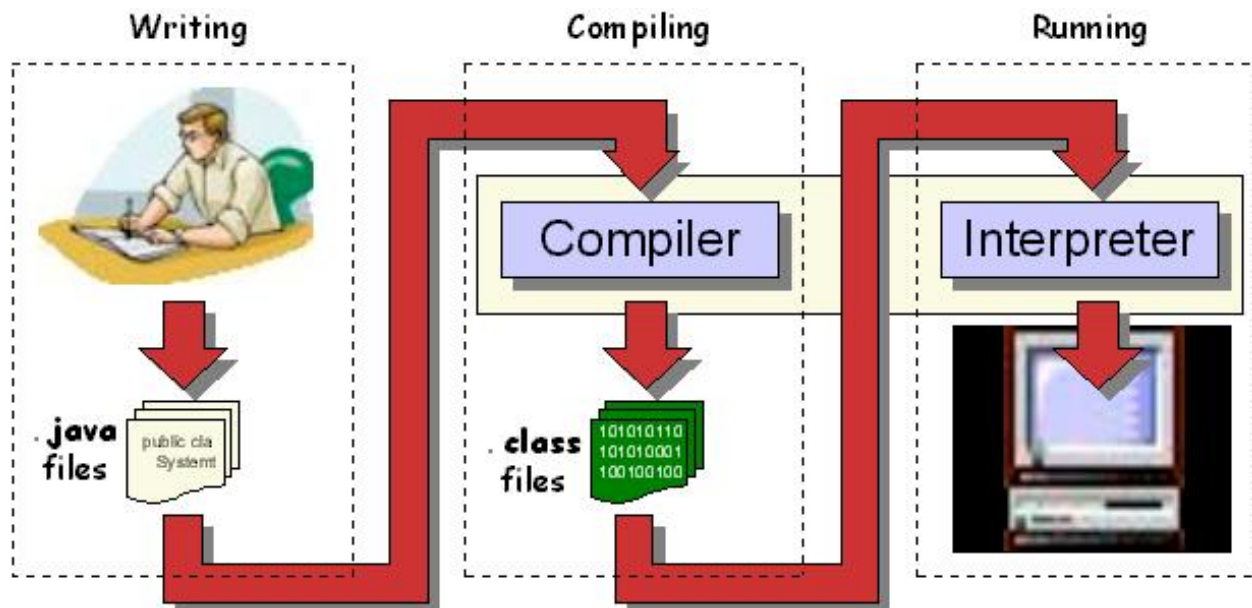
There are many IDE's that you can use. You may choose whatever you wish. Here are a few:

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

1.2 Writing Your First JAVA Program

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

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



The java **compiler**:

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

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

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

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

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

Our First Program

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

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

Here is the first program that we will write:

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

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

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

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

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

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

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

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

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

1.3 Processing vs. Java

Although Processing is based on JAVA syntax, it uses simplified names for functions and procedures. Here is a brief explanation of the differences (and similarities) between the two languages:

Displaying Information To the System Console (need to add `System.out`):

Processing	Java
<pre>print("The avg is " + avg); println("All Done");</pre>	<pre>System.out.print("The avg is " + avg); System.out.println("All Done");</pre>

Math Functions (all in a Math class now):

Processing	Java
<pre> min(a, b) max(a, b) round(a) pow(a, b) sqrt(a) abs(a) sin(a) cos(a) tan(a) degrees(r) radians(d) random(n) dist(x₁, y₁, x₂, y₂) </pre>	<pre> Math.min(a, b) Math.max(a, b) Math.round(a) Math.pow(a, b) Math.sqrt(a) Math.abs(a) Math.sin(a) Math.cos(a) Math.tan(a) Math.toDegrees(r) Math.toRadians(d) Math.random() Point.distance(x₁, y₁, x₂, y₂) </pre>

Commenting Code (remains the same):

Processing	Java
<pre> // print("Avg:" + avg); /* x = x + 2; y = y - 7; println(x + "," + y); */ </pre>	<pre> // System.out.print("Avg:" + avg); /* x = x + 2; y = y - 7; System.out.println(x + "," + y); */ </pre>

Variables and Constants (remains the same ... but floats need *f* afterwards):

Processing	Java
<pre> boolean hungry = true; int days = 15; byte age = 19; short years = 3467; long seconds = 17102394892; char gender = 'M'; float amount = 21.3; double weight = 165.23; final int DAYS = 365; final float RATE = 4.923; final double PI = 3.1415965; </pre>	<pre> boolean hungry = true; int days = 15; byte age = 19; short years = 3467; long seconds = 17102394892; char gender = 'M'; float amount = 21.3f; //need f double weight = 165.23; final int DAYS = 365; final float RATE = 4.923f; //need f final double PI = 3.1415965; </pre>

Type Conversion (is different ... no longer functions, but part of JAVA syntax):

Processing	Java
<pre>double d = 65.237898546; float f = (float)d; // 65.2379 int i = int(f); // 65 float g = float(i); // 65.0 char c = char(i); // A</pre>	<pre>double d = 65.237898546; float f = (float)d; // 65.2379 int i = (int)f; // 65 float g = (float)i; // 65.0 char c = (char)i; // A</pre>

Arrays (remains the same, even for 2D arrays):

Processing	Java
<pre>int[] days; double[] weights; String[] names; Car[] rentals; Person[] friends; days = new int[30]; weights = new double[100]; names = new String[3]; rentals = new Car[500]; friends = new Person[50]; int[] ages = {34, 12, 45}; double[] weights = {4.5, 2.6, 1.5}; String[] names = {"Bill", "Jen"};</pre>	<pre>int[] days; double[] weights; String[] names; Car[] rentals; Person[] friends; days = new int[30]; weights = new double[100]; names = new String[3]; rentals = new Car[500]; friends = new Person[50]; int[] ages = {34, 12, 45}; double[] weights = {4.5, 2.6, 1.5}; String[] names = {"Bill", "Jen"};</pre>

IF statements (remains the same):

Processing	Java
<pre>if ((grade >= 80) && (grade <=100)) println("Super!"); if (grade >= 50) { print(grade); println(" is a pass."); } else println("Grade too low.");</pre>	<pre>if ((grade >= 80) && (grade <=100)) System.out.println("Super!"); if (grade >= 50) { System.out.print(grade); System.out.println(" is a pass."); } else System.out.println("Grade too low.");</pre>

SWITCH statements (remains the same):

Processing	Java
<pre>switch(month) { case 2: print(28); break; case 4: case 6: case 9: case 11: print(30); break; default: print(31); }</pre>	<pre>switch(month) { case 2: System.out.print(28); break; case 4: case 6: case 9: case 11: System.out.print(30); break; default: System.out.print(31); }</pre>

FOR / WHILE loops (remains the same):

Processing	Java
<pre>int total = 0; for (int i=1; i<=n; i++) { total = total + i; } println(total); int speed = 0; int x=0; while (x <= width) { drawCarAt(x); speed = speed + 2; x = x + speed; }</pre>	<pre>int total = 0; for (int i=1; i<=n; i++) { total = total + i; } System.out.println(total); int speed = 0; int x=0; while (x <= width) { drawCarAt(x); speed = speed + 2; x = x + speed; }</pre>

Procedures & Functions (essentially remains the same ... sort of ... as we will soon see):

Processing	Java
<pre>void procName(int x, char c) { // Write code here } double funcName(float h) { // Write code here }</pre>	<pre>void procName(int x, char c) { // Write code here } double funcName(float h) { // Write code here }</pre>

As the course continues, you will notice other differences between Processing and JAVA. Many of the differences are with respect to graphics and event handling. However, the underlying concepts are the same.

1.4 Getting User Input

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

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

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

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

This line of code will wait for the user (i.e., you) to enter some text characters using the keyboard. It actually waits until you press the **Enter** key. Then, it returns to you the characters that you typed (not including the **Enter** key). You can then do something with the characters, such as print them out. Here is a simple program that asks users for their name and then says hello to them:

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

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

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

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

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

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

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

```

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

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

        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter three numbers:");

        sum = keyboard.nextInt() + keyboard.nextInt() + keyboard.nextInt();

        System.out.println("The average of these numbers is " + (sum/3.0));
    }
}

```

Here is the output when the **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

```

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

```

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

```

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

```

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

```

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

1.5 Using Objects in JAVA

Until now, we have discussed creating programs by creating a class and inserting all of our code into a **main()** procedure/method. This means that our programs are considered procedural. **Object-Oriented Programming** (OOP) is *similar* to that of procedural programming in that it also 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).



JAVA actually has a **lot** of pre-defined objects that are all organized into various **packages**. A package is essentially equivalent to a folder that contains your **.java** files. 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:

<code>java.lang</code>	Basic classes and interfaces required by many JAVA programs. It is automatically imported into all programs.
<code>java.util</code>	Utility classes and interfaces such as date/time manipulations, random numbers, string manipulation, collections ...
<code>java.io</code>	Classes that enable programs to input and output data.
<code>java.text</code>	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 so that it knows where to find it:

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

We did this already when we used the **Scanner** class, which is in the **java.util** package. 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.

Here is a simple example that makes use of the pre-defined **Object**, **String**, **Date**, **Point** and **Rectangle** object classes in JAVA, making sure to import the correct package:

```

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

public class ObjectTestProgram {
    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
    }
}

```

If we do not specify where to find the objects via the **import** statement, JAVA will become confused when compiling our code and will generate compile errors such as this:

Error: C:\...\ObjectTestProgram.java:12: cannot find symbol class Date

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. So here is the simplest form of the code:

```

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

public 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
    }
}

```

In this example, we are simply creating the objects and then displaying them. Notice how these objects are displayed in the output:

```

java.lang.Object@141ed7ac

Wed Apr 06 15:18:05 EDT 2011
java.awt.Point[x=50,y=75]
java.awt.Rectangle[x=5,y=10,width=20,height=30]

```

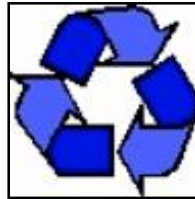
Each object displays itself differently. Notice that the **Date** object that was created actually corresponds to today's date and time (i.e., on April 6, 2011 when I ran the code). Also, notice that the **String** object was actually an empty string (i.e., no characters were displayed).

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

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

Inheritance

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



Encapsulation

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



Polymorphism

- simplifies code understanding
- standardizes method naming



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

In the previous course, we did indeed define our own data structures (a.k.a. *objects*) that we used within our program. Recall these for example:

<pre>class FullName { String firstName; String middleName; String lastName; }</pre>	<pre>class Address { FullName name; int streetNumber; String streetName; String city; String province; String postalCode; }</pre>
<pre>class Employee { boolean employed; boolean hasDegree; int age; int yearsWorked; }</pre>	<pre>class BankAccount { Address owner; int accountNumber; float balance; }</pre>
<pre>class Car { int x; int y; float speed; }</pre>	<pre>class House { int x; int y; float s; }</pre>

Each of these objects were created in the same **Processing** file and we were able to create variables of those types and use them within our program:

<pre>Car myCar; Car yourCar; House aHouse; class Car { ... } class House { ... } void setup() { ... } void draw() { ... }</pre>
--

In JAVA, there are a couple of problems doing this:

1. JAVA requires ALL of our code to be defined **within a class**. So, we cannot define any variables at the top of the program like this.
2. There is no **setup()** or **draw()** procedure in JAVA, a **main()** procedure is used instead.

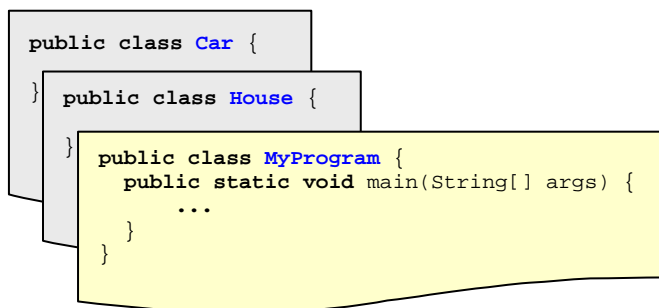
We could therefore write the code inside the **main()** method, making sure to define the classes first as follows:

```
public class MyProgram {
    public static void main(String[] args) {
        class Car {
            ...
        }
        class House {
            ...
        }

        Car myCar;
        Car yourCar;
        House aHouse;

        // Write some code here
    }
}
```

However, it is generally bad programming practice to define multiple classes within another class. In this course, we will be defining ALL of our classes in separate **.java** files which will reside in the same folder as the main program class:



Even though the **Car** and **House** objects are defined in their own individual **.java** files, they cannot be **run** as programs. You can only run classes that have the **public static void main(...)** method defined. So, a java program will typically consist of multiple **.java** files.

As an example, we can define very simple **Car** and **Person** objects and test them as follows:

```
public class Car {
}
```

```
public class Person {
}
```

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

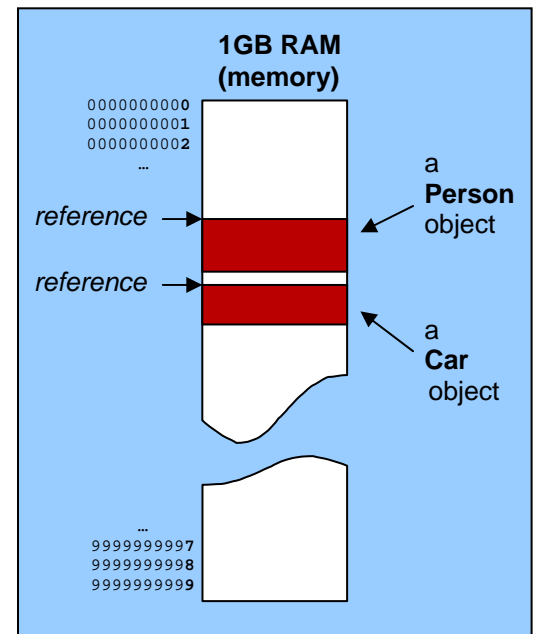
Notice now the output from the program:

```
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.

Recall that 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.



Just to be clear on the terminology, when we save and compile the file **Car.java** and **Person.java**, we are actually creating 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. 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**.