Aminu Rabiu Kiru

COMPUTER PROGRAMMING 1

ATLANTIC INTERNATIONAL UNIVERSITY

1.0 Introduction

Java is a programming language, originally developed by Sun Microsystems and released in 1995. Java applications are typically compiled to bytecode, although compilation to native machine code is also possible. At runtime, bytecode is usually either interpreted or compiled to native code for execution, although direct hardware execution of bytecode by a Java processor is also possible.

The language derives much of its syntax from C and C++, but has a simpler object model and fewer low-level facilities. JavaScript, a scripting language, shares a similar name and has similar syntax, but is not directly related to Java.

The original Java compilers, virtual machines, and class libraries were developed by Sun Microsystems from 1995. Others have also developed alternative implementations of these Sun technologies, the most well known of which are the free software Java implementations, such as the GNU Compiler for Java and GNU Classpath.

Between November 2006 and May 2007, Sun Microsystems made available most of their Java technologies under the GNU General Public License, in compliance with the specifications of the Java Community Process, thus making almost all of Sun's Java also free software.

1.1 Platform independence

Platform independence, means that programs written in Java language must run similarly on any supported hardware operating system platform. One should be able to write a program once, compile it once, and run it anywhere.

This is achieved by most Java compilers by compiling the Java language code halfway to bytecode (specifically Java bytecode), simplified machine instructions specific to the Java platform. The code is then run on a virtual machine (VM), a program written in native code on the host hardware that interprets and executes generic Java bytecode. (In some JVM versions, bytecode can also be compiled to native code, resulting in faster execution.) Further, standardized libraries are provided to allow access to features of the host machines (such as graphics, threading and networking) in unified ways. Note that, although there is an explicit

compiling stage, at some point, the Java bytecode is interpreted or converted to native machine instructions by the JIT compiler.

1.2 The Java Virtual Machine

Machine language consists of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in high-level programming languages such as Java, Pascal, or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called a compiler. A compiler takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language). If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.

There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an interpreter, which translates it instruction by instruction, as necessary. An interpreter is programs that acts much like a CPU, with a kind of fetch and execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine language commands to do so.

One use of interpreters is to execute high-level language programs. For example, the programming language Lisp is usually executed by an interpreter rather than a compiler. However, interpreters have another purpose; they can let you use a machine language program meant for one type of computer on a completely different type of computer. For example, there is a program called Virtual PC that runs on Macintosh computers. Virtual PC is an interpreter that executes machine language programs written for IBM-PC clone computers. If you run Virtual PC on

your Macintosh, you can run any PC program, including programs written for Windows. Unfortunately, a PC program will run much more slowly than it would on an actual IBM clone. The problem is that Virtual PC executes several Macintosh machine language instructions for each PC machine language instruction in the program it is interpreting. Compiled programs are inherently faster than interpreted programs.

The designers of Java chose to use a combination of compilation and interpretation. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called virtual computer is known as the Java virtual machine. The machine language for the Java virtual machine is called Java bytecode. There is no reason why Java bytecode could not be used as the machine language of a real computer, rather than a virtual computer.

However, the main interesting point of Java is that it can actually be used on any computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the Java virtual machine in the same way that Virtual PC simulates a PC computer. A different Jave bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter; it can run any Java bytecode program; and the same Java bytecode program can be run on any computer that has such an interpreter. This is one of the essential features of Java; the same compiled program can be run on many different types of computers as shown in figure 1.2 below.

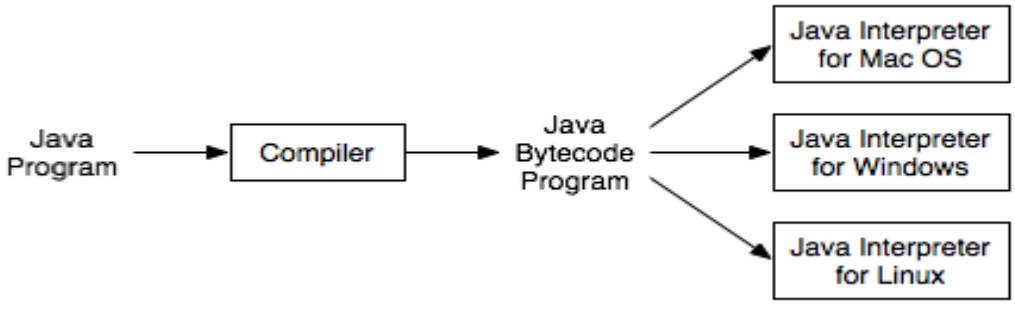

Figure 1.2

2.0 Objects and Object-oriented Programming

Objects are software bundles of data and related methods. Objects are the key concept to understanding object oriented technology, because software objects are modeled after real world objects, you can represent real world objects in programs using software objects. For example, you might want to represent dogs in an animation program or a bicycle in an indoor electronic exercise bike. However, you can also use software objects to objectify abstract concepts. For example, event is a common object used in GUI (**G**raphical **U**ser **I**nterface) window systems to represent the event when a user presses a mouse button or strikes a key on the keyboard.

Figure 2.0 illustrates a common visual representation of a software object:

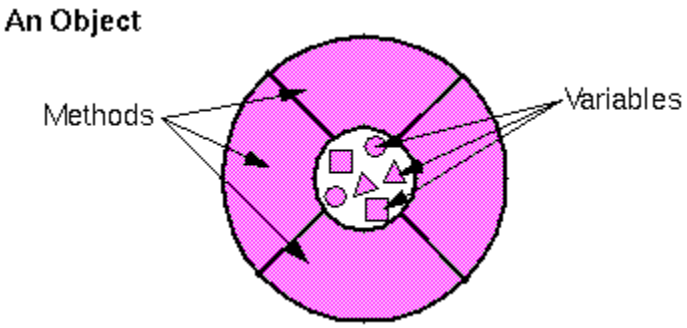

Figure 2.0

Everything that the software object knows (state) and can do (behaviour) is expressed by the variables and methods within that object. A software object that modeled your real world bicycle would have variables that indicated the bicycle's current state; it's moving at 10 mph, its pedal cadence is 90 rpm, and it's in 5th gear.

2.1 Program must be designed

It is not just possible for one to just sit down at the computer and compose a program of any complexity. The discipline called software engineering is concerned with the construction of correct, working, well written programs. The

software engineer tends to use accepted and proven methods for analyzing the problem to be solved and for designing a program to solve that problem.

During the 1970s and into the 80s, the primary software engineering methodology was structured programming. The structured programming approach to program design was based on the following advice:

- To solve a large problem, break the problem into several pieces and work on each piece separately.
- To solve each piece, treat it as a new problem which can itself be broken down into smaller problems eventually, and also can be solve directly without further decomposition. This approach is called top down programming.

Although, there is nothing wrong with top down programming. It is a valuable and often used as an approach to problem solving. However, it is incomplete; because the design of the data structures for a program was as least as important as the design of subroutines and control structures. Top down programming doesn't give adequate consideration to the data that the program manipulates.

Another problem with strict top down programming is that it makes it difficult to reuse work done for other projects. By starting with a particular problem and subdividing it into convenient pieces, top down programming tends to produce a design that is unique to that problem. It is unlikely that you will be able to take a large chunk of programming from another program and fit it into your project, at least not without extensive modification. Producing high quality programs is difficult and expensive, so programmers and the people who employ them are always eager to reuse past work.

Top down design is often combined with bottom up design. In bottom up design, the approach is to start at the bottom, with problems that you already know how to solve and for which you might already have a reusable software component at hand. From there, you can work upwards towards a solution to the overall problem.

The reusable components should be as modular as possible. A module is a component of a larger system that interacts with the rest of the system in a simple, well defined, straight forward manner. The idea is that a module can be plugged into a system. The details of what goes on inside the module are not important to the system as a whole, as long as the module fulfills its assigned role correctly. This is called information hiding, and it is one of the most important principles of software engineering.

One common format for software modules is to contain some data, along with some subroutines for manipulating that data. For example, a mailing-list module might contain a list of names and addresses along with a subroutine for adding a new name, a subroutine for printing mailing labels, and so forth. In such modules, the data itself is often hidden inside the module; a program that uses the module can then manipulate the data only indirectly, by calling the subroutines provided by the module. This protects the data, since it can only be manipulated in known, well defined ways. And it makes it easier for programs to use the module, since they don't have to worry about the details of how the data is represented. Information about the representation of the data is hidden.

Modules that could support this kind of information-hiding became common in programming languages in the early 1980s. Since then, a more advanced form of the same idea has more or less taken over software engineering. This latest approach is called **O**bject **O**riented **P**rogramming, often abbreviated as OOP.

The central concept of object oriented programming is the object, which is a kind of module containing data and subroutines. The point of view in OOP is that an object is a kind of self sufficient entity that has an internal state the data it contains and that can respond to messages (calls to its subroutines). A mailing list object, for example, has a state consisting of a list of names and addresses.

The OOP approach to software engineering is to start by identifying the objects involved in a problem and the messages that those objects should respond to. The program that results is a collection of objects, each with its own data and its own

set of responsibilities. The objects interact by sending messages to each other. There is not much top down in such a program. However, object oriented programs tend to be better models of the way the world itself works, and that they are therefore easier to write, easier to understand, and more likely to be correct.

It is common for objects to bear a kind of family resemblance to one another. Objects that contain the same type of data and that respond to the same messages in the same way belong to the same class. (In actual programming, the class is primary; that is, a class is created and then one or more objects are created using that class as a template). But objects can be similar without being in exactly the same class.

For example, consider a drawing program that lets the user draw lines, rectangles, ovals, polygons, and curves on the screen. In the program, each visible object on the screen could be represented by a software object in the program. There would be five classes of objects in the program, one for each type of visible object that can be drawn. All the lines would belong to one class, all the rectangles to another class, and so on. These classes are obviously related; all of them represent drawable objects. They would, for example, all presumably be able to respond to a draw yourself message. Another level of grouping, based on the data needed to represent each type of object, is less obvious, but would be very useful in a program; we can group polygons and curves together as multipoint objects, while lines, rectangles, and ovals are two point objects. A line is determined by its endpoints, a rectangle by two of its corners, and an oval by two corners of the rectangle that contains it. We could diagram these relationships as illustrated in Figure 2.1

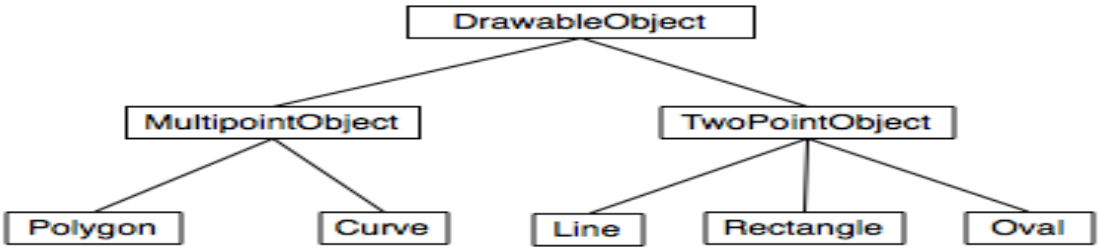

Figure 2.1

DrawableObject, MultipointObject, and TwoPointObject would be classes in the program. MultipointObject and TwoPointObject would be subclasses of DrawableObject. The class Line would be a subclass of TwoPointObject and indirectly of DrawableObject. A subclass of a class is said to inherit the properties of that class. The subclass can add to its inheritance and it can even override part of that inheritance by defining a different response to some method. Nevertheless, lines, rectangles, and so on are drawable objects, and the class DrawableObject expresses this relationship.

Inheritance is a powerful means for organizing a program. It is also related to the problem of reusing software components. A class is the ultimate reusable component. Not only can it be reused directly if it fits exactly into a program you are trying to write, but if it just almost fits, you can still reuse it by defining a subclass and making only the small changes necessary to adapt it exactly to your needs.

So, Object Oriented Program (OOP) is meant to be both a superior program development tool and a partial solution to the software reuse problem.

2.2 Variables and the Primitive Types

In programs, names are fundamentals to programme; names are used to refer to many different sorts of things. In order to use those things, a programmer must understand the rules for giving names to things and the rules for using the names to work with those things. That is, the programmer must understand the syntax and the semantics of names.

According to the syntax rules of Java, a name is a sequence of one or more characters. It must begin with a letter or underscore and must consist entirely of letters, digits, and underscores. For example, here are some legal names:

N n rate x15 quite_a_long_name HelloWorld

No spaces are allowed in identifiers; HelloWorld is a legal identifier, but Hello World is not. Upper case and lower case letters are considered to be different, so that HelloWorld, helloworld, HELLOWORLD, and hElloWorLD are all distinct names. Certain names are reserved for special uses in Java, and cannot be used by the programmer for other purposes. These reserved words include: class, public, static, if, else, while, and several dozen other words.

Java is actually pretty liberal about what counts as a letter or a digit. Java uses the unicode character set, which includes thousands of characters from many different languages and different alphabets, and many of these characters count as letters or digits.

The pragmatics of naming includes style guidelines about how to choose names for things. For example, it is customary for names of classes to begin with upper case letters, while names of variables and of subroutines begin with lower case letters; you can avoid a lot of confusion by following the same convention in your own programs. Most Java programmers do not use underscores in names, although some do use them at the beginning of the names of certain kinds of variables. When a name is made up of several words, such as HelloWorld or interestRate, it is customary to capitalize each word, except possibly the first; this is sometimes referred to as camel case, since the upper case letters in the middle of a name are supposed to look something like the humps on a camel's back.

3.0 Variables

Programs manipulate data that are stored in memory. In machine language, data can only be referred to by giving the numerical address of the location in memory where it is stored. In a high-level language such as Java, names are used instead of numbers to refer to data. It is the job of the computer to keep track of where in memory the data is actually stored; the programmer only has to remember the name. A name used in this way is refers to data stored in memory is called a variable.

A variable is not a name for the data itself but for a location in memory that can hold data. You should think of a variable as a container or box where you can store

data that you will need to use later. The variable refers directly to the box and only indirectly to the data in the box. Since the data in the box can change, a variable can refer to different data values at different times during the execution of the program, but it always refers to the same box.

In Java, the only way to get data into a variable (that is, into the box that the variable names) is with an assignment statement. An assignment statement takes the form:

variable **=** expression**;**

where expression represents anything that refers to or computes a data value. When the computer comes to an assignment statement in the course of executing a program, it evaluates the expression and puts the resulting data value into the variable. For example, consider the simple assignment statement

```
rate = 0.07;
```

The variable in this assignment statement is rate, and the expression is the number 0.07. The computer executes this assignment statement by putting the number 0.07 in the variable rate, replacing what ever was there before. Now, consider the following more complicated assignment statement:

```
interest = rate * principal;
```

Here, the value of the expression ‘rate * principal’ is being assigned to the variable interest. In the expression, the * is a multiplication operator that tells the computer to multiply rate times principal. The names rate and principal are themselves variables, and it is really the values stored in those variables that are to be multiplied. Now it can be seen that when a variable is used in an expression, it is the value stored in the variable that matters; in this case, the variable seems to refer to the data in the box, rather than to the box itself. When the computer executes this assignment statement, it takes the value of rate, multiplies it by the value of principal, and stores the answer in the box referred to by interest. When a

variable is used on the left hand side of an assignment statement, it refers to the box that is named by the variable.

Note that, an assignment statement is a command that is executed by the computer at a certain time. It is not a statement of fact. For example, suppose a program includes the statement ‘rate = 0.07;’ If the statement ‘interest = rate * principal;’ is executed later in the program, can we say that the principal is multiplied by 0.07, the answer is No. The value of rate might have been changed in the mean time by another statement. The meaning of an assignment statement is completely different from the meaning of an equation in mathematics, even though both use the symbol ‘=’.

3.1 Types and Literals

A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if you try to violate this rule; because Java is a strongly typed language.

There are eight so called primitive types built into Java. The primitive types are named byte, short, int, long, float, double, char, and Boolean. The first four types hold integers (whole numbers such as 17, -38477, and 0). The four integer types are distinguished by the ranges of integers they can hold. The float and double types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type character holds a single character from the Unicode character set. And a variable of type Boolean holds one of the two logical values true or false.

Any data value stored in the computer's memory must be represented as a binary number, that is as a string of 0’s and 1’s. A single 0 or 1 is called a bit. A string of eight bits is called a byte. Memory is usually measured in terms of bytes. The byte data type refers to a single byte of memory. A variable of type byte holds a string of eight bits, which can represent any of the integers between -128 and 127, inclusive. (There are 256 integers in that range; eight bits can represent 256 i.e. two raised to the power eight (or 2^8) different values. As for the other integer types,

- short corresponds to two bytes (16 bits). Variables of type short have values in the range -32768 to 32767.
- int corresponds to four bytes (32 bits). Variables of type int have values in the range -2147483648 to 2147483647.
- long corresponds to eight bytes (64 bits). Variables of type long have values in the range -9223372036854775808 to 9223372036854775807.

You don't have to remember these numbers, but they do give you some idea of the size of integers that you can work with. Usually, you should just stick to the int data type, which is good enough for most purposes.

The float data type is represented in four bytes of memory, using a standard method for encoding real numbers. The maximum value for a float is about 10 raised to the power 38 (or 10^{38}). A float can have about 7 significant digits. (So that 32.3989231134 and 32.3989234399 would both have to be rounded off to about 32.398923 in order to be stored in a variable of type float.) A double take up 8 bytes, can range up to about 10 to the power 308(or 10^{308}), and has about 15 significant digits. Ordinarily, double type for real values should be stick to.

A variable of type character occupies two bytes in memory. The value of a character variable is a single character such as A, *, x, or a space character. The value can also be a special character such a tab or a carriage return or one of the many Unicode characters that come from different languages. When a character is typed into a program, it must be surrounded by single quotes; for example: 'A', '*', or 'x'. Without the quotes, A would be an identifier and * would be a multiplication operator. The quotes are not part of the value and are not stored in the variable; they are just a convention for naming a particular character constant in a program.

A name for a constant value is called a literal. A literal is what need to be type in a program to represent a value. 'A' and '*' are literals of type character, representing the character values A and *. Certain special characters have special literals that

use a backslash (\ ), as an escape character. In particular, a tab is represented as '\t', a carriage return as '\r', a line feed as '\n', the single quote character as '\'', and the backslash itself as '\\'. Note that even though you type two characters between the quotes in '\t', the value represented by this literal is a single tab character.

Numeric literals are a little more complicated than you might expect. Of course, there are obvious literals such as 317 and 17.42. But there are other possibilities for expressing numbers in a Java program. First of all, real numbers can be represented in an exponential form such as 1.3e12 or 12.3737e-108. The e12 and e-108 represent powers of 10, so that 1.3e12 means 1.3 times 10^{12} and 12.3737e-108 means 12.3737 times 10^{-108}. This format can be used to express very large and very small numbers. Any numerical literal that contains a decimal point or exponential is a literal of type double. To make a literal of type float, you have to append an F or f to the end of the number. For example, 1.2F stands for 1.2 considered as a value of type float. (Occasionally, you need to know this because the rules of Java say that you can't assign a value of type double to a variable of type float, so you might be confronted with a ridiculous-seeming error message if you try to do something like x = 1.2; when x is a variable of type float. You have to say x = 1.2F;. This is one reason why I advise sticking to type double for real numbers.)

Even for integer literals, there are some complications. Ordinary integers such as 177777 and -32 are literals of type byte, short, or int, depending on their size. You can make a literal of type long by adding L as a suffix. For example: 17L or 728476874368L. As another complication, Java allows octal (base-8) and hexadecimal (base-16) literals. I don't want to cover base-8 and base-16 in detail, but in case you run into them in other people's programs, it's worth knowing a few things: Octal numbers use only the digits 0 through 7. In Java, a numeric literal that begins with a 0 is interpreted as an octal number; for example, the literal 045 represents the number 37, not the number 45. Hexadecimal numbers use 16 digits, the usual digits 0 through 9 and the letters A, B, C, D, E, and F. Upper case and lower case letters can be used interchangeably in this context. The letters

represent the numbers 10 through 15. In Java, a hexadecimal literal begins with 0x or 0X, as in 0x45 or 0xFF7A.

Hexadecimal numbers are also used in character literals to represent arbitrary Unicode characters. A Unicode literal consists of \u followed by four hexadecimal digits. For example, the character literal '\u00E9' represents the Unicode character that is an e with an acute accent.

For the type Boolean, there are precisely two literals: true and false. These literals are typed just as I've written them here, without quotes, but they represent values, not variables. Boolean values occur most often as the values of conditional expressions. For example,

```
rate > 0.05;
```

is a Boolean valued expression that evaluates to true if the value of the variable rate is greater than 0.05, and to false if the value of rate is not greater than 0.05. Boolean values can also be assigned to variables of type Boolean. Java has other types in addition to the primitive types, but all the other types represent objects rather than primitive data values. For the most part, we are not concerned with objects for the time being. However, there is one predefined object type that is very important; the type String. A String is a sequence of characters. You've already seen a string literal “Hello World!”. The double quotes are part of the literal; they have to be typed in the program. However, they are not part of the actual string value, which consists of just the characters between the quotes. Within a string, special characters can be represented using the backslash notation. Within this context, the double quote is itself a special character. For example, to represent the string value

```
I said, "Are you listening!"
```

with a linefeed at the end, you would have to type the string literal:

```
"I said, \"Are you listening!\"\n"
```

You can also use \t, \r, \\, and unicode sequences such as \u00E9 to represent other special characters in string literals. Because strings are objects, their behavior in programs is peculiar in some respects.

3.2 Variables in Programs

A variable can be used in a program only if it has first been declared. A variable declaration statement is used to declare one or more variables and to give them names. When the computer executes a variable declaration, it sets aside memory for the variable and associates the variable's name with that memory. A simple variable declaration takes the form:

type-name variable-name-or-names;

The variable-name-or-names can be a single variable name or a list of variable names separated by commas. (We'll see later that variable declaration statements can actually be somewhat more complicated than this.) Good programming style is to declare only one variable in a declaration statement, unless the variables are closely related in some way. For example:

```
int numberOfStudents;
String name;
double x, y;
boolean isFinished;
char firstInitial, middleInitial, lastInitial;
```

It is also good style to include a comment with each variable declaration to explain its purpose in the program, or to give other information that might be useful to a human reader. For example:

```
double principal;    // Amount of money invested.
double interestRate; // Rate as a decimal, not percentage.
```

Variables declared inside a subroutine are called local variables for that subroutine. They exist only inside the subroutine, while it is running, and are completely inaccessible from outside. Variable declarations can occur anywhere inside the subroutine, as long as each variable is declared before it is used in any expression. Some people like to declare all the variables at the beginning of the subroutine. Others like to wait to declare a variable until it is needed. Preferably, declare important variables at the beginning of the subroutine, and use a comment to explain the purpose of each variable. Declare utility variables which are not important to the overall logic of the subroutine at the point in the subroutine where they are first used. Here is a simple program using some variables and assignment statements:

```
/**
 * This class implements a simple program that
 * will compute the amount of interest that is
 * earned on ₦15, 000 invested at an interest
 * rate of 0.07 for one year.  The interest and
 * the value of the investment after one year are
 * printed to standard output.
 */

public class Interest {

   public static void main(String[] args) {

      /* Declare the variables. */

      double principal;     // The value of the investment.
      double rate;          // The annual interest rate.
      double interest;      // Interest earned in one year.
```

```
        /* Do the computations. */

        principal = 15000;
        rate = 0.07;
        interest = principal * rate;    // Compute the interest.

        principal = principal + interest;
              // Compute value of investment after one year, with interest.
              // (Note: The new value replaces the old value of principal.)

        /* Output the results. */

        System.out.print("The interest earned is $");
        System.out.println(interest);
        System.out.print("The value of the investment after one year is $");
        System.out.println(principal);

    } // end of main()
       } // end of class Interest
```

This program uses several subroutine call statements to display information to the user of the program. Two different subroutines are used: System.out.print and System.out.println. The difference between these is that System.out.println adds a linefeed after the end of the information that it displays, while System.out.print does not. Thus, the value of interest, which is displayed by the subroutine call System.out.println(interest); follows on the same line after the string displayed by the previous System.out.print statement. Note that the value to be displayed by System.out.print or System.out.println is provided in parentheses after the subroutine name. This value is called a parameter to the subroutine. A parameter provides a subroutine with information it needs to perform its task. In a subroutine call statement, any parameters are listed in parentheses after the subroutine

name. Not all subroutines have parameters. If there are no parameters in a subroutine call statement, the subroutine name must be followed by an empty pair of parentheses.

3.3 A Class

A class is a template or prototype that defines the variables and the methods common to all objects of a certain kind.

In the real world, you often have many objects of the same kind. For example, your bicycle is really just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle is an instance of the class of objects knows as bicycles. All bicycles have some states (current gear, current cadence, two wheels) and behaviors (change gears, brake) in common.

When building bicycles, manufacturers take advantage of the fact that bicycles share characteristics and they build many bicycles from the same blueprint, it would be very inefficient to produce a new blueprint for every individual bicycle they manufactured.

In object-oriented software, it's also possible to have many objects of the same kind that share characteristics like rectangles, employee records, video clips and so on. Like the bicycle manufacturers, you can take advantage of the fact that objects of the same kind share certain characteristics and you can create a blueprint for those objects. Software blueprints for objects are called classes as illustrated in figure 3.3 below.

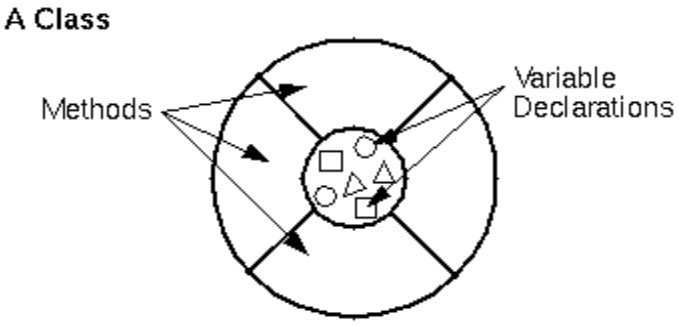

Figure 3.3 below

For example, you could create the bicycle class that would declare several variables to contain the current gear, the current cadence, etc. It would also declare and provide implementations for the methods that allow the rider to change gears, brake and change the pedaling cadence as shown in figure 3.3a below.

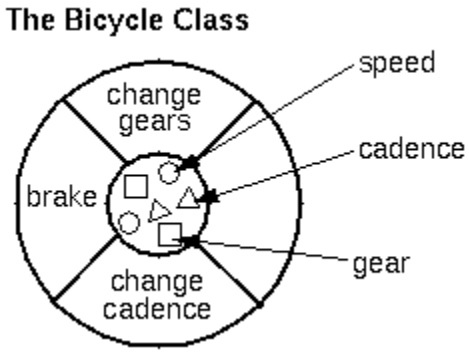

Figure 3.3a

The values for the variables are provided by each instance of the class. So, after you've created the bicycle class, you must create an instance of it to use it. When you create an instance from a class, the variables declared by the class are allocated in memory. Then you can use the instance's methods to assign values to the variables. Instances of the same class share method implementations.

3.4 Defining a class

```
import java.util.Date;
class DateApp {
   public static void main(String args[]) {
      Date today = new Date();
      System.out.println(today);
   }
}
```

The bold line in the DateApp application listing above begins a class definition block. A class (the basic building block of an object-oriented language such as the Java language) is a template that describes the data and behaviour associated with instances of that class. When you instantiate a class you create an object that will look and feel like other instances of the same class. The data associated with a

class or object is called variables; the behaviour associated with class or object are called methods.

A traditional example of a class from the world of programming is a class that represents a rectangle. The class would contain variables for the origin of the rectangle, and its width and height. The class would also contain a method that calculates the area of the rectangle. An instance of the rectangle class would contain the information for a specific rectangle: like the dimensions of the floor of your office, or the dimensions of the window you are using to view this page.

In the Java language, the general form of a class definition is

```
class name {
    . . .
}
```

where the keyword class begins the class definition for a class named name. The variables and methods of the class are embraced by the curly brackets that begin and end the class definition block. DateApp has no variables and has a single method named main().

3.4.1 A Simple Class

```
class Count {
   public static void main(String args[])
      throws java.io.IOException
                {
      int count = 0;

      while (System.in.read() != -1)
         count++;
      System.out.println("Input has " + count + " chars.");
   }
}
```

In Java language, all methods and variables must exist within a class. So, the first line of the character counting application defines a class, Count, that defines the

methods, variables, and any other classes needed to implement the character counting application. Since this program is such a simple one, the Count class just defines one method named main().

3.4.2 The Benefit of Classes

Objects provide the benefit of modularity and information hiding. Classes provide the benefit of reusability. Bicycle manufacturers reuse the same blueprint over and over again to build lots of bicycles. Software programmers use the same class over and over to again create many objects.

4.0 Selection

A Selection implements a Selector for use with a Parameter or Value wherever selective comparison of Parameters and/or Values is needed. For example, both the Parameter and Value Find methods use a Selector when making object comparisons, and use this Selection class when no other Selector is specified. A Selection can be used to specify the criteria to be used in comparing all, or any part, of two Parameters or Values.

The criteria are specified as the characteristics of the Parameter and/or Value to use in making a selection, and the combinatorial boolean logic to apply to the characteristics when comparing one object against another. The Selector interface defines the criteria symbols. The Selection class provides the definition of the criteria and comparison methods.

4.1 The switch Statement

Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with enumerated types and a few special classes that "wrap" certain primitive types: Character, Byte, Short, and Integer.

The following program, SwitchDemo1, declares an int named month whose value represents a month out of the year. The program displays the name of the month, based on the value of month, using the switch statement.

```
class SwitchDemo {
    public static void main(String[] args) {
        int month = 8;
        switch (month) {
            case 1:  System.out.println("January"); break;
            case 2:  System.out.println("February"); break;
            case 3:  System.out.println("March"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("May"); break;
            case 6:  System.out.println("June"); break;
            case 7:  System.out.println("July"); break;
            case 8:  System.out.println("August"); break;
            case 9:  System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Invalid month.");break;
        }
    }
}
```

In this case, "August" is printed to standard output.
The body of a switch statement is known as a switch block. Any statement immediately contained by the switch block may be labeled with one or more case or default labels. The switch statement evaluates its expression and executes the appropriate case.
Of course, you could also implement the same thing with if-then-else statements:

```
int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
. . . // and so on
```

Deciding whether to use if-then-else statements or a switch statement is sometimes a judgment call. You can decide which one to use based on readability and other factors. An if-then-else statement can be used to make decisions based on ranges of values or conditions, whereas a switch statement can make decisions based on only a single integer or enumerated value.
Another point of interest is the break statement after each case. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, case statements fall through; that is, without an explicit break, control will flow sequentially through subsequent case statements. The following program, SwitchDemo2, illustrates why it might be useful to have case statements fall through:

```
class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1:
```

```
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
               numDays = 31;
               break;
            case 4:
            case 6:
            case 9:
            case 11:
               numDays = 30;
               break;
            case 2:
               if ( ((year % 4 == 0) && !(year % 100 == 0))
                    || (year % 400 == 0) )
                  numDays = 29;
               else
                  numDays = 28;
               break;
            default:
               System.out.println("Invalid month.");
               break;
         }
         System.out.println("Number of Days = " + numDays);
      }
}
```

This is the output from the program.

Number of Days = 29

Technically, the final break is not required because flow would fall out of the switch statement anyway. However, using a break is recommend, so that modifying the code is easier and less error-prone. The default section handles all values that aren't explicitly handled by one of the case sections.

5.0 Repetition

Also called iteration or looping (if else is sometimes known as branching). Although the if statement, it is known as a < conditional > statement since the next statement to be executed depends upon the condition in the if part.

5.1 The for Statement

The ‘for’ statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the ‘for’ statement can be expressed as follows:

```
for (initialization; termination; increment) {
    statement(s)
}
```

When using this version of the ‘for’ statement, keep in mind that; the initialization expression initializes the loop; it's executed once, as the loop begins. When the termination expression evaluates to false, the loop terminates. The increment expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.

The following program, ForDemo, uses the general form of the for statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
```

```
                System.out.println("Count is: " + i);
            }
        }
    }
```

The output of this program is:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the ‘for’ statement, so it can be used in the termination and increment expressions as well. If the variable that controls a ‘for’ statement is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names i, j, and k are often used to control for loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the ‘for’ loop is optional; an infinite loop can be created as follows:

```
for ( ; ; ) {    // infinite loop

    // your code goes here
}
```

The ‘for’ statement also has another form designed for iteration through collections and arrays. This form is some times referred to as the enhanced ‘for’ statement,

and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[ ] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, EnhancedForDemo, uses the enhanced for to loop through the array:

```
class EnhancedForDemo {
    public static void main(String[] args){
        int[] numbers = {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
          System.out.println("Count is: " + item);
        }
    }
}
```

In this example, the variable item holds the current value from the numbers array. The output from this program is the same as before:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

This form of the ‘for’ statement is recommended instead of the general form whenever possible.

5.2 The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {
    statement(s)
}
```

The ‘while’ statement evaluates expression, which must return a boolean value. If the expression evaluates to true, the ‘while’ statement executes the statement(s) in the ‘while’ block. The ‘while’ statement continues testing the expression and executing its block until the expression evaluates to false. Using the ‘while’ statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

You can implement an infinite loop using the ‘while’ statement as follows:

```
while (true){
   // your code goes here
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
    statement(s)
} while (expression);
```

The difference between ‘do-while’ and ‘while’ is that ‘do-while’ evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count <= 11);
    }
}
```

6.0 Coding, Testing and Debugging

It would be nice if, having developed an algorithm for your program; you could relax, press a button, and get a perfectly working program. Unfortunately, the process of turning an algorithm into Java source code doesn't always go smoothly. And when you do get to the stage of a working program, it's often only working in the sense that it does something. Unfortunately not what you want it to do.

After program design comes coding, that is translating the design into a program written in Java or some other language. Usually, no matter how careful you are, a few syntax errors will creep in from somewhere, and the Java compiler will reject your program with some kind of error message. Unfortunately, while a compiler will always detect syntax errors, it's not very good about telling you exactly what's wrong. Sometimes, it's not even good about telling you where the real error is. A spelling error or missing "{" on line 45 might cause the compiler to choke on line

105. You can avoid lots of errors by making sure that you really understand the syntax rules of the language and by following some basic programming guidelines. For example, never type a "{" without typing the matching "}". Then go back and fill in the statements between the braces. A missing or extra brace can be one of the hardest errors to find in a large program. Always, indent your program nicely. If you change the program, change the indentation to match. It's worth the trouble. Use a consistent naming scheme, so you don't have to struggle to remember whether you called that variable interestrate or interestRate. In general, when the compiler gives multiple error messages, don't try to fix the second error message from the compiler until you've fixed the first one. Once the compiler hits an error in your program, it can get confused, and the rest of the error messages might just be guesses. May be the best advice is; take the time to understand the error before you try to fix it. Programming is not an experimental science, it is the real thing.

When your program compiles without error, you are still not done. You have to test the program to make sure it works correctly. Remember that the goal is not to get the right output for the two sample inputs. The goal is a program that will work correctly for all reasonable inputs. Ideally, when faced with an unreasonable input, it will respond by gently chiding the user rather than by crashing. Test your program on a wide variety of inputs. Try to find a set of inputs that will test the full range of functionality that you've coded into your program. As you begin writing larger programs, write them in stages and test each stage along the way. You might even have to write some extra code to do the testing, for example, to call a subroutine that you've just written. You don't want to be faced, if you can avoid it, with 500 newly written lines of code that have an error in there some where.

The point of testing is to find bugs semantic errors that show up as incorrect behavior rather than as compilation errors. And the sad fact is that you will probably find them. Again, you can minimize bugs by careful design and careful coding, but no one has found a way to avoid them altogether. Once you've detected a bug, it's time for debugging. You have to track down the cause of the

bug in the program's source code and eliminate it. Debugging is a skill that, like other aspects of programming, requires practice to master. So one essential debugging skill is the ability to read source code the ability to put aside preconceptions about what you think it does and to follow it the way the computer does mechanically, step-by-step to see what it really does. This is very hard and difficult to do.

Often, it's a problem just to find the part of the program that contains the error. Most programming environments come with a debugger, which is a program that can help you find bugs. Typically, your program can be run under the control of the debugger. The debugger allows you to set "breakpoints" in your program. A breakpoint is a point in the program where the debugger will pause the program so you can look at the values of the program's variables. The idea is to track down exactly when things start to go wrong during the program's execution. The debugger will also let you execute your program one line at a time, so that you can watch what happens in detail once you know the general area in the program where the bug is lurking.

A more traditional approach to debugging is to insert debugging statements into your program. These are output statements that print out information about the state of the program. Typically, a debugging statement would say something like

```
System.out.println("At start of while loop, N = "+ N);
```

You need to be able to tell from the output where in your program the output is coming from, and you want to know the value of important variables. Sometimes, you will find that the computer isn't even getting to a part of the program that you think it should be executing. Remember that the goal is to find the first point in the program where the state is not what you expect it to be. That's where the bug is.

And finally, remember the golden rule of debugging: If you are absolutely sure that everything in your program is right, and if it still doesn't work, then one of the things that you are absolutely sure of is wrong.

7.0 Constructors

Objects are created with the operator, new. For example, a program that wants to use a PairOfDice object could say:

```
PairOfDice dice;   // Declare a variable of type PairOfDice.

dice = new PairOfDice();  // Construct a new object and store a
                          //   reference to it in the variable.
```

In this example, "new PairOfDice()" is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object. This reference is the value of the expression, and that value is stored by the assignment statement in the variable, dice, so that after the assignment statement is executed, dice refers to the newly created object. Part of this expression, "PairOfDice()", looks like a subroutine call, and that is no accident. It is, in fact, a call to a special type of subroutine called a constructor. This might puzzle you, since there is no such subroutine in the class definition. However, every class has at least one constructor. If the programmer doesn't write a constructor definition in a class, then the system will provide a default constructor for that class. This default constructor does nothing beyond the basics, allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other subroutine, with three exceptions. A constructor does not have any return type (not even void). The name of the constructor must be the same as the name of the class in which it is defined. The only modifiers that can be used on a constructor definition are the access modifiers public, private, and protected. (In particular, a constructor can't be declared static.)

However, a constructor does have a subroutine body of the usual form, a block of statements. There are no restrictions on what statements can be used. And it can

have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the PairOfDice class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:

```
public class PairOfDice {

   public int die1;   // Number showing on the first die.
   public int die2;   // Number showing on the second die.

   public PairOfDice(int val1, int val2) {
         // Constructor.  Creates a pair of dice that
         // are initially showing the values val1 and val2.
      die1 = val1;  // Assign specified values
      die2 = val2;  //           to the instance variables.
   }

   public void roll() {
         // Roll the dice by setting each of the dice to be
         // a random number between 1 and 6.
      die1 = (int)(Math.random()*6) + 1;
      die2 = (int)(Math.random()*6) + 1;
   }

} // end class PairOfDice
```

The constructor is declared as "public PairOfDice(int val1, int val2) ...", with no return type and with the same name as the name of the class. This is how the Java compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression "new PairOfDice(3,4)" would create a PairOfDice object in

which the values of the instance variables die1 and die2 are initially 3 and 4. In a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice;           // Declare a variable of type PairOfDice.

dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice
                            //   object that initially shows 1, 1.
```

Now that the constructor is added to the PairOfDice class, we can no longer create an object by saying "new PairOfDice()"! The system provides a default constructor for a class only if the class definition does not already include a constructor, so there is only one constructor in the class, and it requires two actual parameters. However, this is not a big problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the PairOfDice class, it might have a constructor with no parameters which produces a pair of dice showing random numbers:

```
public class PairOfDice {

   public int die1;   // Number showing on the first die.
   public int die2;   // Number showing on the second die.

   public PairOfDice() {
         // Constructor.  Rolls the dice, so that they initially
         // show some random values.
      roll();  // Call the roll() method to roll the dice.
   }

   public PairOfDice(int val1, int val2) {
         // Constructor.  Creates a pair of dice that
         // are initially showing the values val1 and val2.
```

```
        die1 = val1;  // Assign specified values
        die2 = val2;  //           to the instance variables.
    }

    public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Now that the option of constructing a PairOfDice object either with "new PairOfDice()" or with "new PairOfDice(x,y)", where x and y are int-valued expressions.

This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation "(int)(Math.random()*6)+1", because it's done inside the PairOfDice class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the PairOfDice class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value. This illustrates once again that you can create several instances of the same class:

```
public class RollTwoPairs {

    public static void main(String[] args) {

        PairOfDice firstDice;  // Refers to the first pair of dice.
        firstDice = new PairOfDice();
```

```
PairOfDice secondDice; // Refers to the second pair of dice.
secondDice = new PairOfDice();

int countRolls;  // Counts how many times the two pairs of
                 //    dice have been rolled.

int total1;      // Total showing on first pair of dice.
int total2;      // Total showing on second pair of dice.

countRolls = 0;

do {  // Roll the two pairs of dice until totals are the same.

   firstDice.roll();    // Roll the first pair of dice.
   total1 = firstDice.die1 + firstDice.die2;   // Get total.
   System.out.println("First pair comes up  " + total1);

   secondDice.roll();    // Roll the second pair of dice.
   total2 = secondDice.die1 + secondDice.die2;   // Get total.
   System.out.println("Second pair comes up " + total2);

   countRolls++;   // Count this roll.

   System.out.println();  // Blank line.

} while (total1 != total2);

System.out.println("It took " + countRolls
                + " rolls until the totals were the same.");
```

```
    } // end main()

} // end class RollTwoPairs
```

Constructors are subroutines, but they are subroutines of a special type. They are certainly not instance methods, since they don't belong to objects. Since they are responsible for creating objects, they exist before any object is created. They are more like static member subroutines, but they are not and cannot be declared to be static. In fact, according to the Java language specification, they are technically not members of the class at all! In particular, constructors are not referred to as “methods.”

Unlike other subroutines, a constructor can only be called using the new operator, in an expression that has the form

new class-name **(** parameter-list **)**

where the parameter-list is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a subroutine call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created.

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.
2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed

and stored in the instance variable. Otherwise, the default initial value is used.

3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.
4. The statements in the body of the constructor, if any, are executed.
5. A reference to the object is returned as the value of the constructor call.

The end result of this is that you have a reference to a newly constructed object. You can use this reference to get at the instance variables in that object or to call its instance methods.

8.0 Arrays

An array is group of variables that is given only one name. Each variable that makes up the array is given a number instead of a name which makes it easier to work with using loops among other things.

8.1 Declaring an array

Declaring an array is the same as declaring a normal variable except that you must put a set of square brackets after the variable type. Here is an example of how to declare an array of integers.

```
public class Array
{
  public static void main(String[] args)
  {
    int[ ] a;
  }
}
```

An array is more complex than a normal variable so we have to assign memory to the array when we declare it. When you assign memory to an array you also set its size. Here is an example of how to create an array that has 5 elements.

```
public class Array
{
   public static void main(String[] args)
   {
      int[ ] a = new int[5];
   }
}
```

Instead of assigning memory to the array you can assign values to it instead. This is called initializing the array because it is giving the array initial values.

```
public class Array
{
   public static void main(String[] args)
   {
      int[] a = {12, 23, 34, 45, 56};
   }
}
```

8.2 Using an array

The values in an array are accessible using the number of the element you want to access between square brackets after the array's name. There is one important thing that must be remember about arrays which is they always start at 0 and not 1. Here is an example of how to set the values for an array of 5 elements.

```
public class Array
{
   public static void main(String[] args)
   {
      int[ ] a = new int[5];
      a[0] = 12;
      a[1] = 23;
      a[2] = 34;
      a[3] = 45;
      a[4] = 56;
```

```
  }
}
```

A much more useful way of using an array is in a loop. Here is an example of how to use a loop to set all the values of an array to 0 which you will see is much easier than setting all the values to 0 seperately.

```
public class Array
{
  public static void main(String[] args)
  {
    int[] a = new int[5];
    for (int i = 0; i < 5; i++)
      a[i] = 0;
  }
}
```

8.3 Sorting an array

Sometimes you will want to sort the elements of an array so that they go from the lowest value to the highest value or the other way around. To do this we must use the bubble sort. A bubble sort uses an outer loop to go from the last element of the array to the first and an inner loop which goes from the first to the last. Each value is compared inside the inner loop against the value in front of it in the array and if it is greater than that value then it is swapped. Here is an example.

```
public class Array
{
  public static void main(String[ ] args)
  {
    int[] a = {3, 5, 1, 2, 4};
    int i, j, temp;
    for (i = 4; i >= 0; i--)
      for (j = 0; j < i; j++)
        if (a[j] > a[j + 1])
```

```
        {
          temp = a[j];
          a[j] = a[j + 1];
          a[j + 1] = temp;
        }
    }
}
```

8.3.1 2D arrays

So far we have been using 1-dimensional or 1D arrays. A 2D array can have values that go not only down but also across. Here are some figures that will explain the difference between the 2 (i.e. 1D and 2D arrays) in a better way.

1D Array

0	1
1	2
2	3
3	4
4	5

2D Array

	0	**1**	**2**
0	1	2	3
1	4	5	6
2	7	8	9

All that you need to do to create and use a 2D array is use 2 square brackets instead of 1.

```
public class Array
{
   public static void main(String[ ] args)
   {
      int[ ] [ ] a = new int[3] [3];
```

```
    a[0] [0] = 1;
  }
}
```

References:

1. David J. Eck
   Introduction to Programming Using Java, Fifth Edition
   Version 5.0.1 MAY 2007 **www.oopweb.com** JULY 2007

2. Ken Arnold, James Gosling, David Holmes by Ken Arnold, James Gosling, David Holmes
   The Java(TM) Programming Language (3rd Edition) **www.abebooks.com** JULY 2007

3. Jeff Friesen, Object-oriented language basics, Part 1
   04/06/01 www.javaWorld.com JULY 2007

4. Kasper B. Graversen
   OOP in Java Repetition Constants Selection Exception. **www.it-cdk** JULY 2007