Chapter One

Introduction to Object Oriented Programming (OOP in Java)

What is Object-Oriented Programming?

Object Oriented Programming (OOP) is a different method of thinking about programming using object and classes. We **think separately** about **data** and how the **methods** interact with data in non-OOP. Object-Oriented Programming, however, forces us to think in terms of objects and the interaction between objects. An object is a self-contained entity that describes not only certain data, but also the methods to maintain the data.

An object-oriented approach identifies the keywords in the problem. The keywords would be the **object** in the implementation and the hierarchy defines the relationship between **these objects**. The term object is used here to describe a limited well-defined structure, **containing all the information about some entity - data type and methods to manipulate the data.**

Why do we use Object-Oriented Programming?

OOP enable us to model real-world problem through programs in more logical manner **-** Because objects are discrete entities, we can debug, modify and maintain them more easily **-** If our objects are thoughtfully designed, we can **reuse** much of our code than is the case with procedural programming. OOP is the most usable and maintainable programming due to the following basic features.

Basic Concept/features of OOP

Abstraction

Abstraction is the way of collecting relevant information (attribute and methods) from the existing problem.

- Abstraction: The act of identifying objects to model the problem domain.
- Classes are abstracted from concepts/Real world problem.
- This is the first step of identifying **classes** and **attributes** that will be used in your applications.
- Objects are instantiated from classes.

Encapsulation

A goal of OOP is to differentiate the use of an object from the implementation of that object. One method to accomplish this is through the usage of data binding/hiding. Data hiding enable us to completely encapsulate an object's data members.

The data-binding paradigm does not allow non-member methods to access data members directly. It is a good programming practice to make our entire data members private while defining a good interface for the object.

Encapsulation is the grouping of related items into one unit.

- Attributes and behaviors /methods are encapsulated to create objects.
- Implementation details are hidden from the outside world.
- The packaging of operations and attributes representing state into an object type so that state is **accessible** or **modifiable only through the objects' interface.**

Inheritance

Inheritance is a technique for creating a new class (subclass) from an existing class (superclass) by adding more functionality to it. We say that the new class inherits all the functionality from the existing class.

- A subclass is derived from a superclass. Example: An `Employee` is a `Person`.
- The subclass inherits the attributes and behavior of the superclass.
- The subclass can override the behavior of the superclass.
- Inheritance supports code re-use.

Polymorphism

The term polymorphism is derived from a Greek term meaning **many form use of a single method**.

- A method can have many different forms of behavior.
- A single method may be defined upon more than one class and may take on different implementations in each of those classes.

Message passing

- Objects communicate by sending messages.
- Messages convey some form of information.
- An object requests another object to carry out an activity by sending it a message.
- Most messages pass arguments back and forth through the object using **dot operator**.

Classes

A class is **user-defined data type** that is used to implement an abstract object, giving us the capability to use OOP. A class includes members. A member can be either data known as a data member or a method, known as a member method.

Objects

An object is an example /instance of a class. An object includes all the data necessary to represent the item and the methods that manipulate the data. Objects can be uniquely identified by its name and it defines a state that is represented by the values of its attribute at a particular point of time.

Members of a class

A class has one or more variable types, called members. The two types of members are *data members* and *methods member*.

Data members: data members of a class are exactly variables.

Methods member: are methods/ functions defined within a class that act on the data members in the class.

```
Example:
class student {
        String Stud_name;  // data members
        String Stud_IDNo;
        int Age;
 public static void main(String[] args) { // Main function
      student stud;
      stud.Stud_name=”Abel”;
      stud.Age=24;
System.out.println(stud.Stud_name); //methods member
System.out.println(stud.Age);
System.out.println(“This is your  first java  code!”);
   } }
```

Assignment

1. What is Programming language, what is the use of programming languages? Give some examples of programming language and classify them as OOP and non-OOP language.
2. Discuss the advantage of object-oriented programming language (OOP) over non-OOP languages (Unstructured and Structured Programming language).

What is Java?

Java is a computer programming language created by Sun Microsystems. Java is used mainly on the Internet and uses a virtual machine which has been implemented in most browsers to translate Java code into a specific application on different computer system. With most programming languages, you either compile or interpret a program so that you can run it on your computer. The Java programming language is unusual in that a program is both compiled and interpreted. The most common Java programs are applications and applets. Applications are standalone programs. Applets are similar to applications, but they don't run standalone. Instead, applets adhere to a set of conventions that lets them run within a Java-compatible browser. Applets are web based applications.

The Java programming language is a high-level language that can be described as,

- Object oriented
- Portable
- Distributed
- Multithreaded

What is JDK?

The **Java Development Kit** (**JDK**) is a Sun Microsystems product aimed at Java developers. The JDK has as its primary components a collection of programming tools, including:

- java – the loader for Java applications. This tool is an interpreter and can interpret the class files generated by the javac compiler. Now a single launcher is used for both development and deployment. The old deployment launcher, jre, no longer comes with Sun JDK, and instead it has been replaced by this new java loader.
- javac – the compiler, which converts source code into Java bytecode
- appletviewer – this tool can be used to run and debug Java applets without a web browser
- javadoc – the documentation generator, which automatically generates documentation from source code comments
- jar – the archiver, which packages related class libraries into a single JAR file. This tool also helps manage JAR files.
- javaws – the Java Web Start launcher for JNLP applications
- jdb – the debugger
- jstat – Java Virtual Machine statistics monitoring tool (experimental)

What does the "Java Virtual Machine (JVM)" mean?

The Java Virtual machine (JVM) is the application that executes a Java program and it is included in the Java package.

A JVM is distributed along with a set of standard class libraries that implement the Java application programming interface (API). Appropriate APIs bundled together form the Java Runtime Environment (JRE).

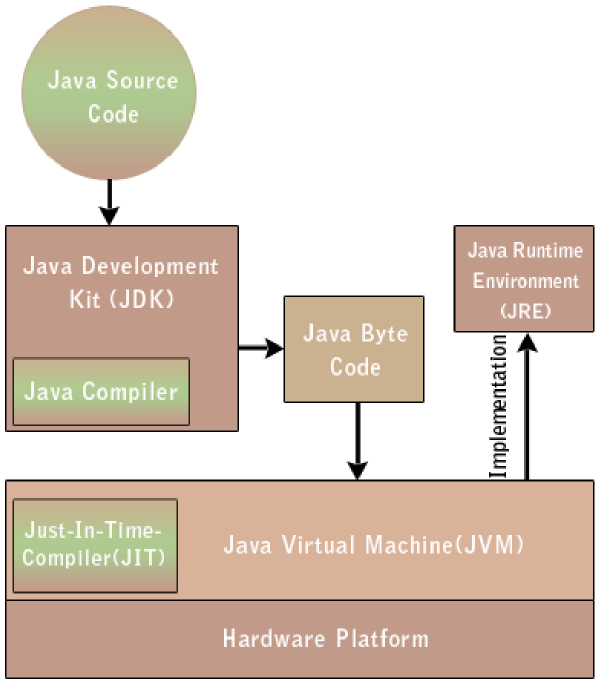
JVMs are available for many hardware and software platforms. The use of the same bytecode for all JVMs on all platforms allows Java to be described as a "compile once, run anywhere" programming language, as opposed to "write once, compile anywhere", which describes cross-platform compiled languages. Thus, the JVM is a crucial component of the Java platform.

As long as a computer has a Java VM (Virtual Machine), a Java program can run on these machines,

- Windows 2000
- Linux
- Solaris
- MacOS

Difference between JVM (Java Virtual Machine) and JDK (Java Development Kit)

The Java Virtual Machine (JVM) executes the Java programs bytecode (.class file). The bytecode is generated after the compilation of the program by the Java compiler. The Java Virtual Machine is the software program and data structure that is on the top of the hardware. The Java virtual machine is called "virtual" because it is an abstract computer (that runs compiled programs) defined by a specification. The Java Virtual Machine is the abstraction between the compiled Java program and used hardware and operating system.


Java Source Code
Java Development Kit (JDK)
Java Compiler
Java Byte Code
Java Runtime Environment (JRE)
Implementation
Just-In-Time-Compiler(JIT)
Java Virtual Machine(JVM)
Hardware Platform

The JDK (Java Development Kit) is used for developing java applications. The JDK includes JRE, set of API classes, Java compiler, Webstart and additional files needed to write Java applications. The JDK (Java Development Kit) contains software development tools which are used to compile and run the Java program. Both JDK and JRE contain the JVM.

Java Virtual Machine (JVM) is an abstract computing machine. Java Runtime Environment (JRE) is an implementation of the JVM. Java Development Kit (JDK) contains JRE along with various development tools like Java libraries, Java source compilers, Java debuggers, bundling and deployment tools.

JVM becomes an instance of JRE at runtime of a java program. It is widely known as a runtime interpreter. The Java virtual machine (JVM) is the cornerstone on top of which the Java technology is built upon. It is the component of the Java technology responsible for its hardware and platform independence.

In general java is:

- An OOP that uses class and objects.
- A very portable object-oriented programming language
- A large supporting class library that covers many general needs.
- Can create Applets, Applications, JavaServer Pages, and more.
- An open standard - the language specification is publicly, freely available.

Your First Java program and program components:

Here is a Java program that displays the message “Hello World!”

```
// A program to display the message
// Hello World! on standard output
public class HelloWorld {
    public static void main(String[] args) {
      System.out.println("Hello World!");
   }
}   // end of class HelloWorld
```

The command that actually displays the message is:

```
System.out.println("Hello World!");
```

This command is an example of a subroutine call statement. It uses a "built-in subroutine/method" named **System.out.println( )** to do the actual work. Recall that a subroutine consists of the instructions for performing some task, chunked together and given a name. That name can be used to "call" the subroutine whenever that task needs to be performed. A built-in subroutine is one that is already defined as part of the language and therefore is automatically available for use in any program.

- When you run this program, the message "Hello World!" (Without the quotes) will be displayed on standard output. The computer will type the output from the program, Hello World!
- You must be curious about all the other stuff in the above program. Part of it consists of comments. Comments in a program are entirely ignored by the computer; they are there for human readers only. This doesn't mean that they are unimportant. Programs are meant to be read by people as well as by computers, and without comments; a program can be very difficult to understand. Java has two types of comments. The first type, used in the above program, begins with // and extends to the end of a line. The computer ignores the // and everything that follows it on the same line. Java has another style of comment that can extend over many lines. That type of comment begins with /* and ends with */.
- Everything else in the program is required by the rules of Java syntax. All programming in Java is done inside "**classes**." The first line in the above program (not counting the comments) says that this is a class named HelloWorld. "HelloWorld," the name of the class, also serves as the name of the program. In order to define a program, a class must include a subroutine named **main( )**, with a definition that takes the form:

```
public static void main(String[] args) {
      //statements
}
```

When you tell the Java interpreter to run the program, the interpreter calls the **main( )** subroutine, and the statements that it contains are executed. These statements make up the script that tells the computer exactly what to do when the program is executed. The main() routine can call subroutines that are defined in the same class or even in other classes, but it is the main() routine that determines how and in what order the other subroutines are used.

The word "public" in the first line of main() means that this routine can be called from **outside the program**. This is essential because the main() routine is called by the Java interpreter, which is something external to the program itself. ***Static*** indicates that main() is a class method not an instance method ,it is static.

The definition of the subroutine -- that is, the instructions that say what it does -- consists of the sequence of "statements" enclosed between braces, { }. A program is defined by a public class that takes the form:

```
public class program-name {
    optional-variable-declarations-and-subroutines/methods
     public static void main(String[] args) {
      // statements
    }
     optional-variable-declarations-and-subroutines/methods
}
```

Variables

What is variable in the concept of programming language?

Programs manipulate data that are stored in memory. In a high-level language such as Java, **names** are used 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 to refer to data stored in memory is called a variable.**

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 are with an assignment statement. An assignment statement takes the form:

```
datatype variablename = value;//variable declaration
```

Data type is an identifier of the variable that can tell the type of data that the variable can hold.

The Java programming language defines the following kinds of variables:

Class Variables (Static Fields)

Variable that holds data that will be shared among all instances of a class. These variables declared with static keyword. A class variable is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence regardless of how many times the class has been instantiated. The code static int numcol = 6; would create such a static field.

Instance Variables (Non-Static Fields)

Instance variables hold data for an instance of a class. Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the static keyword. Non-static fields are also known as **instance** variables because their values are unique to each instance of a class (to each object, in other words)

Local Variables

Local variables are variables that used within a block of codes. Similar to how an object stores its state in fields, a method will often store its temporary state in local variables. The syntax for declaring a local variable is similar to declaring a field (Ex, int count = 0 ;). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared, which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

Naming of variables

Names are fundamental to programming. In programs, 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.

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. We say that Java is a strongly typed language because it enforces this rule.

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

- ✓ Variable names are case-sensitive.
- ✓ A variable's name can be any legal identifier, an unlimited-length sequence of Unicode letters and digits, beginning with a letter.

- ✓ When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting. Also keep in mind that the name you choose must not be a keyword or reserved word like class, main, static, void etc.
- ✓ If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word.

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:

```
   datatype-name variable-name;
```

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;//literal
```

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.
```

Primitive Data Types

The Java programming language is statically-typed, which means that all **variables must first be declared before they can be used.** This involves stating the variable's type and name, as you've already seen:

```
      int product= 1; int sum=0;
```

Doing so tells your program that a field named "product” and “sum” exist and hold numerical data, and have an initial value of "1" and “0” respectively. A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other primitive data types. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

- **byte**: The **byte** data type is an **8-bit** signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters.
- **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with int, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.

- **int**: The int data type is a **32-bit** signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something else. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use long instead.

- **long**: The long data type is a **64-bit** signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you **need a range of values wider** than those provided by int.

- **float**: The float data type is a single-precision **32-bit** floating point. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point.

- **double**: The double data type is a **double-precision 64-bit** floating point. For decimal values, this data type is generally the default choice.

- **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents **one bit** (1 or 0) of information, but its "size" isn't something that's precisely defined.

- **char**: The char data type is a **single 16-bit Unicode character**.

In addition to the eight primitive data types listed above, the Java programming language also provides special support for **character strings** via the java.lang.String class. Enclosing your character string within double quotes will automatically create a new String object.

The Java programming language also supports a few special escape sequences for char and String literals: \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \" (double quote), \' (single quote), and \\ (backslash).

There's also a special null literal that can be used as a value for any reference type. null may be assigned to any variable, except variables of primitive types. There's little you can do with a null value beyond testing for its presence. Therefore, null is often used in programs as a marker to indicate that some object is unavailable.

Arrays

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. This section discusses arrays in greater detail.

The following statement allocates enough memory for arrayOfInts to contain ten integer elements.

```
int[] arrayOfInts = new int[10]
```

In general, when creating an array, you use the new operator, plus the data type of the array elements, plus the number of elements desired enclosed within square brackets ('[' and ']').

```
elementType[] arrayName = new elementType[arraySize]
```

Now that some memory has been allocated for your array, you can assign values to its elements and retrieve those values:

```
for (int j = 0; j < arrayOfInts.length; j ++) {
   arrayOfInts[j] = j;
   System.out.println("[j] = " + arrayOfInts[j]); }
```

Finally, you can use the built-in length property to determine the size of any array. The code

```
System.out.println(anArray.length);// will print the array's size to standard
output.
```

Each item in an array is called an element, and each element is accessed by its numerical index. Numbering /indexing begin with 0.

Creating, Initializing, and Accessing an Array

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as type [], where type is the data type of the contained elements; the square brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). As with variables of other types, the declaration does not actually create an array, it simply tells the compiler that this variable will hold an array of the specified type.

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for ten integer elements and assigns the array to the anArray variable.

```
anArray = new int[10];  // create an array of integers
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // etc.
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
```

Here the length of the array is determined by the number of values provided between { and }.

You can also declare an array of arrays (also known as a multidimensional array) by using two or more sets of square brackets, such as String[][] names. Here String is the data type and names is the name of multidirectional arrays. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is simply an array whose components are themselves arrays. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimArrayDemo program:

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},
                            {"Smith", "Jones"}};
        System.out.println(names[0][0] + names[1][0]); //Mr. Smith
        System.out.println(names[0][2] + names[1][1]); //Ms. Jones
    }
}
```

Strings

A sequence of character data is called a **string** and is implemented in the Java environment by the **String class** (a member of the java.lang package

Java provides a String class to deal with sequences of characters.

String Studentname = "Keria Mohammed";

The String class provides a variety of methods to operate on String objects.

The **equals()** method is used to compare Strings.

The **length()** method returns the number of characters in the String.

Java lets you to concatenate strings together easily using the **+** operator.

Example:

```
class student{
     string firstname=”Abel”;
     String middlename=”Degu”;
  Public static void main(String[] args)  {
      Student stud1=new Student();
    System.out.println(firstname + “  ” + middlename);
    System.out.println(firstname.length( ));
  }   }
```

Control Statements

The control statements are used to control the flow of execution of the program. This execution order depends on the supplied data values and the conditional logic. Java contains the following types of control statements:

1- Selection Statements
2- Repetition Statements
3- Branching Statements like break, continue, return (Read from books / internet)

Selection statements:

1. If Statement

This is a control statement to execute a single statement or a block of code, when the given condition is true and if it is false then it skips **if** block and rest code of program is executed.

Syntax:

```
        if(conditional_expression){
            <statements>;
            ...;
            ...;
    }
```

Example: If n%2 evaluates to 0 then the "if" block is executed. Here it evaluates to 0 so if block is executed. Hence **"This is even number"** is printed on the screen.

```
         int n = 10;
          if(n%2 = = 0){
          System.out.println("This is even number");
        }
```

2. If-else Statement

The **"if-else"** statement is an extension of if statement that provides another option when 'if' statement evaluates to "false" i.e. else block is executed if **"if"** statement is false.

Syntax:

```
if(conditional_expression){
    <statements>;
    ...;
    ...;
```

```
     }
    else{
        <statements>;
        ....;
        ....;
    }
```

Example: If n%2 doesn't evaluate to 0 then else block is executed. Here n%2 evaluates to 1 that is not equal to 0 so else block is executed. So **"This is not even number"** is printed on the screen.

```
                    int n = 11;
                     if(n%2 = = 0){
                    System.out.println("This is even number");
                         }
                  else{
                      System.out.println("This is not even number");     }
```

The following program, IfElseDemo, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseDemo {
    public static void main(String[] args) {
        int testscore = 76;
        char grade;
        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }   }
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

3. Switch Statement

This is an easier implementation to the if-else statements. The keyword **"switch"** is followed by an expression that should evaluates to byte, short, char or int primitive data types, only. In a switch block there can be one or more **labeled cases**. The expression that creates labels for the case must be unique. The switch expression is matched with each case label. Only the matched case is executed, if no case matches then the default statement (if present) is executed.

Syntax:

```
switch(control_expression){
    case expression 1:
        <statement>;
    case expression 2:
        <statement>;
      ...
      ...
    case expression n:
        <statement>;
    default:
        <statement>;
}//end switch
```

Example: Here expression "day" in switch statement evaluates to 5 which matches with a case labeled "5" so code in case 5 is executed that results to output **"Friday"** on the screen.

```
int day = 5;
switch (day) {
       case 1:
            System.out.println("Monday");
            break;
       case 2:
           System.out.println("Tuesday");
           break;
```

```
      case 3:
          System.out.println("Wednesday");
          break;
      case 4:
          System.out.println("Thrusday");
          break;
      case 5:
         System.out.println("Friday");
         break;
      case 6:
         System.out.println("Saturday");
         break;
      case 7:
          System.out.println("Sunday");
          break;
      default:
           System.out.println("Invalid
entry");
           break;
}
```

The following code example, SwitchDemo, declares an int named month whose value represents a month. The code displays the name of the month, based on the value of month, using the switch statement.

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

The body of a switch statement is known as a switch block. A statement in the switch block can be labeled with one or more case or default labels. The switch statement evaluates its expression, and then executes all statements that follow the matching case label.

✍ *You could also display the name of the month with if-then-else statements:*

Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing. An if-then-else statement can test expressions based on ranges of values or conditions, whereas switch statement tests expressions based only on a single integer, enumerated value, or String object.

Another point of interest is the break statement. 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, statements in switch blocks fall through: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered.

Repetition Statements:

1. While loop statements:

This is a looping or repeating statement. It executes a block of code or statements till the given conditions are true. The expression must be evaluated to a boolean value. It continues testing the condition and executes the block of code. When the expression results to false control comes out of loop.

Syntax:

```
while(expression){
    <statement>;
    ...;
    ...;
}
```

Example: Here expression i<=10 is the condition which is checked before entering into the loop statements. When i is greater than value 10 control comes out of loop and next statement is executed. So here i contains value "1" which is less than number "10" so control goes inside of the loop and prints current value of i and increments value of i. Now again control comes back to the loop and condition is checked. This procedure continues until i becomes greater than value "10". So this loop prints values 1 to 10 on the screen.

```
int i = 1;
  //print 1 to 10
    while (i <= 10){
  System.out.println("Num " + i);
         i++;     }
```

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++;
        }    }  }
```

2. Do-while Statements

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);
     }}
```

3. 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 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 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 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 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 are 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 arrays .This form is sometimes 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 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.