# Unit-1

## 1. Introduction to Object Oriented Programming

Object-oriented programming is one of the most interesting and useful inventions in the software development. It has strong roots in programming paradigms and practices. It addresses the problems commonly known as ***software crisis***. The term software crisis describes the software failure in terms of the following:

- Exceeding the software budget
- Software not meeting the user requirements
- Bugs in the software

OOP is programming which deals with building the programs and applications using the objects. These objects are similar to the real-world objects. Every object has an ***identity***, ***attributes*** and ***behavior***. The features of OOP such as ***abstraction***, ***encapsulation***, ***inheritance*** and ***polymorphism*** are closely related to real-world objects.

## 2. Procedural Programming Language Vs Object Oriented Programming language.

There are two main programming paradigms: procedural programming and object-oriented.

### 2.1 Procedural programming language

In procedural programming languages such as C, PASCAL and FOTRAN, a program is a list of instructions. All these instructions are executed in sequence. As the program size increases, the complexity also increases. Hence, it is difficult to understand and maintain. There may be number of variables in the program. The change of a variable somewhere in the program may produce side affect.

There are two main problems: 1) the functions have ***access*** to the global data, hence there is no data security and 2) functions provide poor ***mapping*** of real-world objects. To address and manage this complexity, second programming paradigm was designed.

### 2.2 Object-oriented programming language

In real-world we deal with objects such as car, plane, and bike. Each object has its own Identity (name), attributes (data) and behavior (functions). The attributes define data for an object. For example the attributes of student object are name, roll number, and branch. The behavior defines the functions that operate on the data. There can be one or more functions.

**Difference between procedural and object-oriented programming languages**

| Sl.No | Procedural Programming language | Object-oriented programming language |
|-------|---------------------------------|--------------------------------------|
| 1 | This follows a top-down approach. | OOP takes a bottom-up approach in designing a program. |
| 2 | Program is divided into small chunks based on the functions. | Program is divided into objects depending on the problem. |
| 3 | Separates the data from functions that operate data. | Encapsulates data and methods in a class |
| 4 | Debugging is difficult | Debugging is easier |
| 5 | Each function contains different data. | Each object controls its own data. |
| 6 | Follows a systematic approach to solve the problem. | Focuses on security of the data irrespective of the algorithm. |
| 7 | Different parts of a program are interconnected via parameter passing. | The functions of the objects are linked via message passing. |
| 8 | No easy way for data hiding. | Data hiding is possible in OOP. |
| 9 | No such concept of inheritance in procedural programming. | Inheritance is allowed in OOP. |
| 10 | Operator overloading is not allowed. | Operator overloading is allowed. |
| 11 | C,Pascal, and Fortran. | C++, Java, and  Python |

## 3. The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism.

**Encapsulation**

*Encapsulation* is the mechanism that binds data and methods together into a single unit. It is also known as *data hiding*. It prevents outside code from accessing the data. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. In Java, the basis

of encapsulation is the **class**. A *class* is defined as blue print or prototype or template of an object. The objects are defined as ***instances of a class***. Thus, a class is a **logical** construct; an object has **physical** reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called ***members*** of the class. Specifically, the data defined by the class are referred to as ***member variables* or *instance variables***. The code that operates on that data is referred to as ***member methods* or just *methods***. The members can be **public or private**. When a member is made public any code outside the class can access them. If the members are declared, then only the members of that class can access its members.
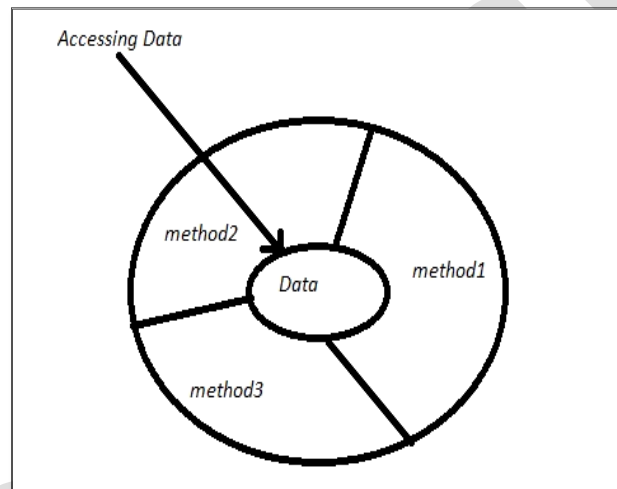


## Inheritance

***Inheritance*** is the process by which one class acquires the properties of another class. The class that acquires properties is called ***sub class***. The class from which properties are acquired is called ***super class***. The relationship between these two classes is called parent-child relationship. When a class inherits another class, it has all the public properties of base class and it adds its own new properties. The main objective of the Inheritance is ***code reusability***.

For example, a vehicle can be car, bike and plane. All vehicles acquire properties from a base class vehicle. All vehicles acquire common properties such as engine and speedometer from the vehicle super class.

**Note:-**  The sub class is also known as ***child*** class or ***derived*** class. The super class is also known as ***parent*** class or ***base*** class.

## Polymorphism

*Polymorphism* simply means "many forms". It can be defined as same thing being used in many different forms. There are two types of polymorphisms: compile-time and run-time polymorphism. In the case of compile-time polymorphism, the compiler will decide which method is going to be executed. The binding is done at compile-time, hence it is called **static binding** or **early binding**.

In case of run-time polymorphism, which method has to be executed is determined at run-time, hence it is called **dynamic binding** or **late binding**. Method overloading is the example for compile-time polymorphism. Method overriding is the example for run-time polymorphism.

## 4. Applications of OOP

The OOP is used in different applications such as Real-time systems, Neural networks, expert systems, database management systems, and artificial intelligence. There are mainly 6 types of applications that can be created using java programming as listed in the Figure 1.
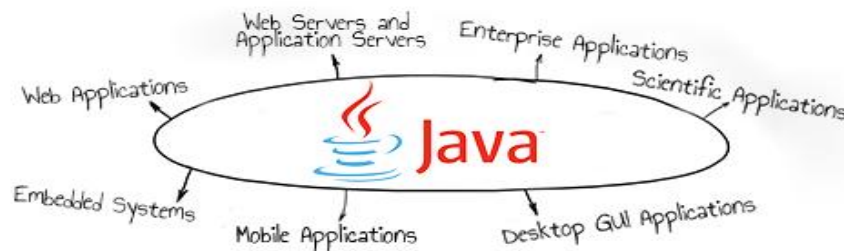


**Figure 1 Applications of OOP**

### 1) Standalone Applications or Desktop GUI Applications

Java provides GUI development through various means like Abstract Windowing Toolkit (AWT), Swing and JavaFX. While AWT contains a number of pre-constructed components such as menu, button, list, and numerous third-party components, Swing, a GUI widget toolkit, additionally provides certain advanced components like trees, tables, scroll panes, tabbed panel and lists.

### 2) Web Application

Java provides support for web applications through Servlets, Struts or JSPs. The easy programming and higher security offered by the programming language has allowed a large number of government applications for health, social security, education and insurance to be based on Java.

### 3. Embedded Systems

Embedded systems, ranging from tiny chips to specialized computers, are components of larger electromechanical systems performing dedicated tasks. Several devices, such as SIM cards, blue-ray disk players, utility meters and televisions, use embedded Java technologies. According to Oracle, 100% of Blue-ray Disc Players and 125 million TV devices employ Java.

### 4) Enterprise Application

An application that is distributed in nature such as banking applications etc;, It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

### 5) Mobile Application

Java Platform, Micro Edition (Java ME or J2ME) is a cross-platform framework to build applications that run across all Java supported devices, including feature phones and smart phones.

### 6) Scientific Applications

Java is the choice of many software developers for writing applications involving scientific calculations and mathematical operations. These programs are generally considered to be fast and secure, have a higher degree of portability and low maintenance. Applications like MATLAB use Java both for interacting user interface and as part of the core system.

## 5. History of JAVA
1. Brief history of Java
2. Java Version History

**Java history** is interesting to know. The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc. For the green team members, it was an

advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape. Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

1) **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

**Why Oak name for java language?**

5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania etc. During 1991 to 1995 many people around the world contributed to the growth of the Oak, by adding the new features. **Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm** were key contributors to the original prototype.

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

**Why Java name for java language?**

7) **Why they chose java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. According to James Gosling "Java was one of the top choices along with **Silk**". Since java was so unique, most of the team members preferred java.

8) Java is an island of Indonesia where **first coffee** was produced (called java coffee).

9) Notice that Java is just a name not an acronym.

10) Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 released in(January 23, 1996).

**Java Version History**

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)

4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)

## 6. Features of Java ( Buzzwords of Java)

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

### Simple

Java was designed to be easy for the professional programmer to learn and use effectively. According to Sun, Java language is simple because:  syntax is based on C++ (so easier for programmers to learn it after C++). removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc. No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

### Secure

Once the byte code generated, the code can be transmitted to other computer in the world with knowing the internal details of the source code.

### Portable

The byte code can be easily carried from one machine to other machine.

### Object Oriented

Everything in Java is an Object. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

**Robust**

The multi-plat-formed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. Java also frees from having worry about many errors. Java is Robust in terms of memory management and mishandled exceptions. Java provides automatic memory management and also provides well defined exception handling mechanism.

**Multithreaded**

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

**Architecture-neutral**

The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

**Interpreted and High Performance**

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java byte code. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java byte code was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

**Distributed**

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

**Dynamic**
Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner.

## 7. Java Virtual Machine



Figure 2 JVM
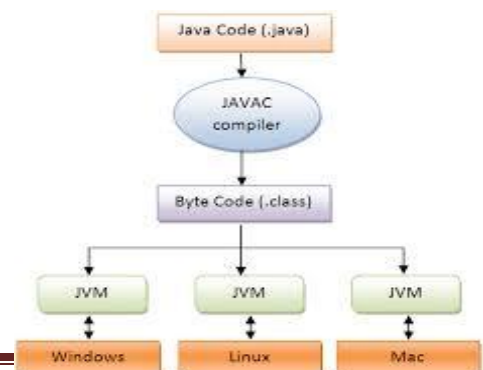
The key that allows Java to solve both the security and the portability problems is the ***byte code***. The output of Java Compiler is not directly executable file. Rather, it contains ***highly optimized*** set of instructions. This set of instructions is called, "***byte code***". This byte code is designed to be executed by Java Virtual Machine **(JVM)**. The JVM also called the interpreter for byte code. JVM also helps to solve many problems associated with web-based programs.

Translating a Java program into byte code makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java byte code. Thus, the execution of byte code by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because byte code has been highly optimized, the use of byte code enables the JVM to execute programs much faster than you might expect

To give on-the-fly performance, the Sun began to design HotSpot Technology for Compiler, which is called, Just-In-Time compiler. The JIT, Compiler also produces output immediately after compilation.

## 8. Program Structures

Simple Java Program

<div align="center">Example.java</div>

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

### Entering the Program

We can use any text editor such as "notepad' or "dos text editor". The source code is typed and is saved with ".java" as extension. The source code contains one or more class definitions. The

program name will be same as class name in which main function is written. This in not compulsory, but by convention this is used. The source file is officially called as compilation unit. We can even uses our choice of interest name for the program. If we use a different name than the class name, then compilation is done with program name, and running is done with class file name. To avoid this confusion and organize the programs well, it is suggested to put the same name for the program and class name, but not compulsory.

**Compiling the Program**

To compile the program, first execute the compiler, "**javac**", specifying the name of the source file on the command line, as shown bellow:

**c:\>javac Example.java**

The javac compiler creates the file called "**Example.class**", that contains the byte code version of the source code. This byte code is the intermediate representation of the source code that contains the instructions that the Java Virtual Machine (JVM) will execute. Thus the output of the javac is not the directly executable code.

   To actually run the program, we must use Java interpreter, called "java". This is interpreter the "**Example.class**" file given as input.

When the program is run with java interpreter, the following output is produced:

---

Hello World

---

**Description of the every line of the program**

The first line contains the keyword **class** and **class name**, which actually the basic unit for encapsulation, in which data and methods are declared.

Second line contains "{" which indicates the beginning of the class.

Third line contains the
**public static void main(String args[])**

where "**public**" is access specifier, when a member of a class is made public it can be accessed by the outside code also. The "**static**" is the keyword which is used to call the main function without instantiation. The "**void**" is also a keyword which is used to describe the the return type of the function. The main function is the beginning of from where execution starts. Java is case-sensitive. "**Main**" is different from the "**main**". In main there is one parameter, String args, which is used to read the command line arguments.

Fourth line contains the "{", which is the beginning of the main function.

Fifth line contains the statement

 **System.out.println("Hello World");**

Here "**System**" is the predefined class, that provides access to the system, and **out** is the output stream that is used to connect to the console. The **println**(), is used to display string passed to it. This can even display other information to.

## Installation of JDK 1.6  ( Additional Topic )
**Installing the JDK Software**

If you do not already have the JDK software installed or if JAVA_HOME is not set, the Java CAPS installation will not be successful. The following tasks provide the information you need to install JDK software and set JAVA_HOME on UNIX or Windows systems.

The following list provides the Java CAPS JDK requirements by platform.
**Solaris**
      JDK5: At least release 1.5.0_14
      JDK6: At least release 1.6.0_03
**IBM AIX**
      JDK5: The latest 1.5 release supported by IBM AIX
**Linux (Red Hat and SUSE)**
      JDK5: At least release 1.5.0_14
      JDK6: At least release 1.6.0_03
**Macintosh**
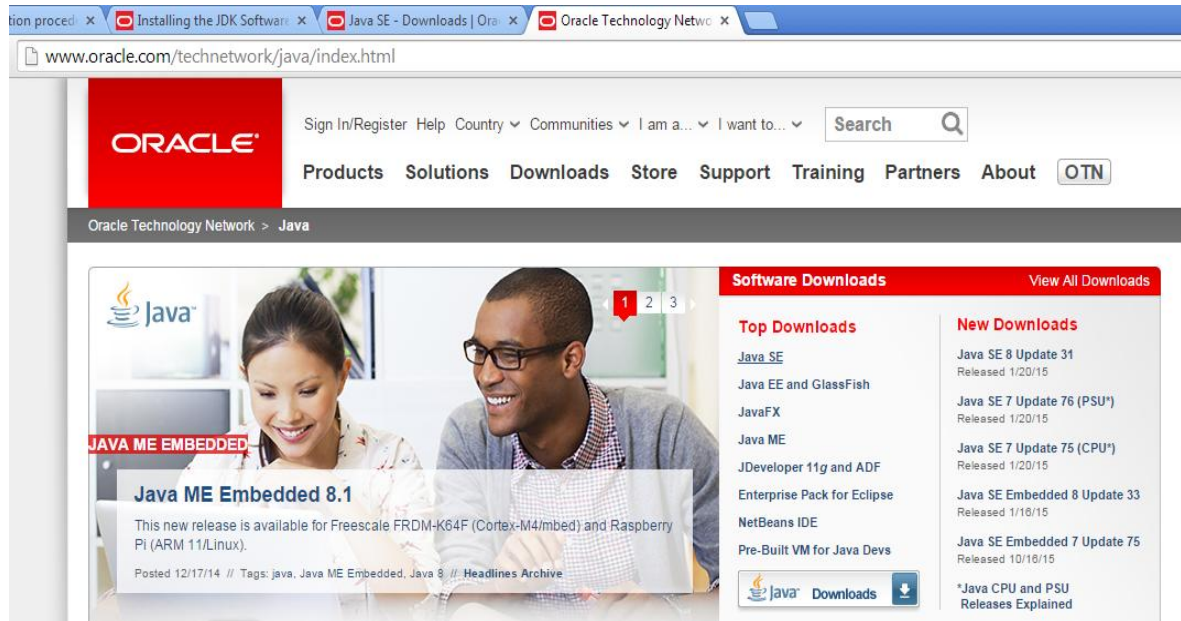      JDK5: The latest 1.5 release supported by Apple
**Microsoft Windows**
      JDK5: At least release 1.5.0_14
      JDK6: At least release 1.6.0_03
## To Install the JDK Software and Set `JAVA_HOME` on a Windows System

1.  Install the JDK software.
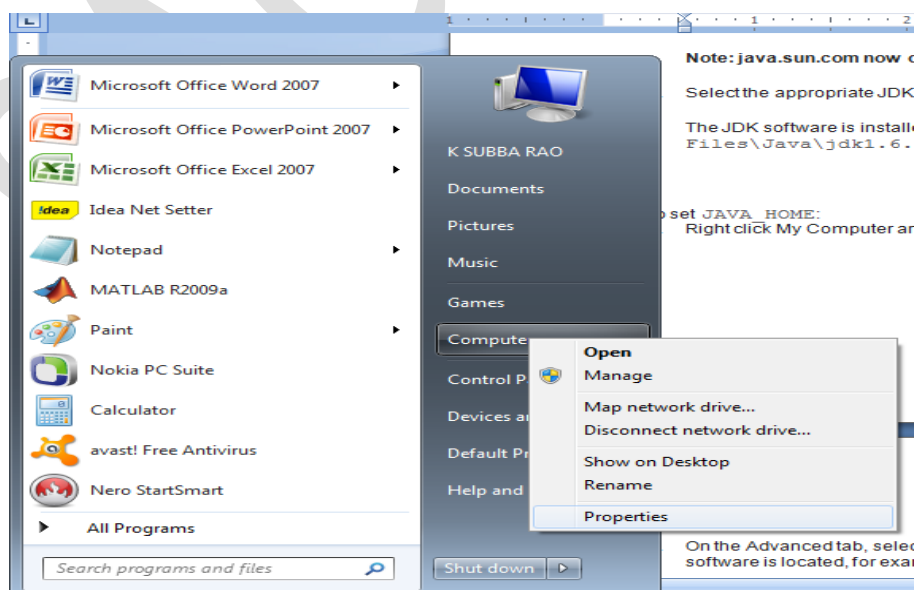    a.  Go to http://java.sun.com/javase/downloads/index.jsp.

**Note: java.sun.com now owned by oracle corporation**

b. Select the appropriate JDK software and click Download.

The JDK software is installed on your computer, for example, at C:\Program Files\Java\jdk1.6.0_02. You can move the JDK software to another location if desired.

2. To set JAVA_HOME:
   a. Right click My Computer and select Properties.



---

b. On the Advanced tab, select Environment Variables, and then edit JAVA_HOME to point to where the JDK software is located, for example, C:\Program Files\Java\jdk1.6.0_02.
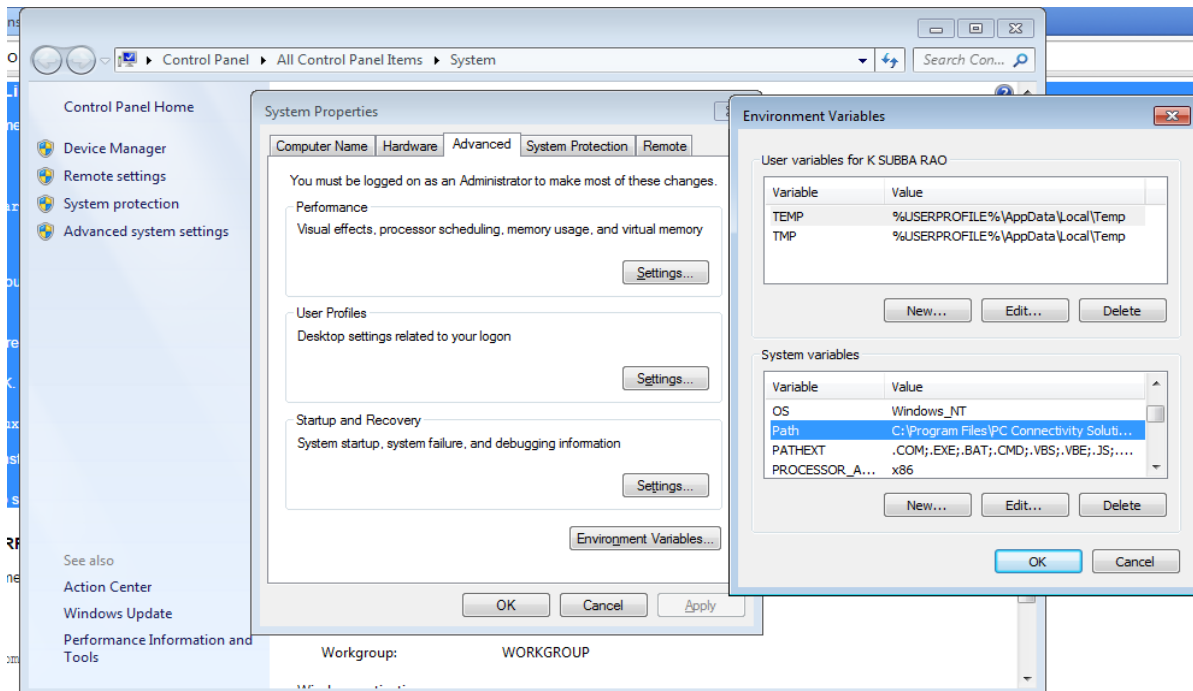


## Installation of the 32-bit JDK on Linux Platforms

This procedure installs the Java Development Kit (JDK) for 32-bit Linux, using an archive binary file (.tar.gz).

These instructions use the following file:

> jdk-8u*version*-linux-i586.tar.gz

1. Download the file.

   Before the file can be downloaded, you must accept the license agreement. The archive binary can be installed by anyone (not only root users), in any location that you can write to. However, only the root user can install the JDK into the system location.

2. Change directory to the location where you would like the JDK to be installed, then move the .tar.gz archive binary to the current directory.

3. Unpack the tarball and install the JDK.

4. % tar zxvf jdk-8u*version*-linux-i586.tar.gz

The Java Development Kit files are installed in a directory called jdk1.8.0_version in the current directory.

5. Delete the .tar.gz file if you want to save disk space.

# Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an *identifier*, *a type*, and an *optional initializer*. In addition, all variables have a scope, which defines their visibility, and a lifetime.

**Declaring a Variable**

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

*type identifier* [ = *value/literal*][, *identifier* [= *value/literal*] ...] ;

Here the type is any primitive data types, or class name. The identifier is the name of the variable. We can initialize the variable by specifying the equal sign and value.
Example

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints, initializing
// d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x';     //the variable x ahs the value 'x'
```

**Dynamic Initialization of the variable**
We can also assign a value to the variable dynamically as follow:
```
int x=12;
int y=13;
float z=Math.sqrt(x+y);
```

**The Scope and Lifetime of Variables**
- ✓ Java allows, to declare a variable within any block.
- ✓ A block begins with opening curly brace and ended with end curly brace.
- ✓ Thus, each time we start new block, we create new scope.
- ✓  A scope determines what objects are visible to parts of your program. It also determines the life time of the objects.
- ✓ Many programming languages define two scopes: **Local and Global**
- ✓ As a general rule a variable defined within one scope, is not visible to code defined outside of the scope.

✓ Scopes can be also nested. The variable defined in **outer scope** are visible to the **inner scopes**, but reverse is not possible.

**Example code**
```
void function1()
{//outer block
        int a;
//here a, are visible to the inner scope
int a=10;
if(a==10)
{// inner block
        int b=a*20;
        int c=a+30;
}//end of inner block
b=20*2;
// b is not known here, which declared in inner scope
}//end of the outer block
```

# Identifiers

Identifiers are used for **class names, method names, and variable names**. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of valid identifiers are:

### Rules for Naming Identifier:
1. *The first character of an identifier must be a letter, or dollar($) sign.*
2. *The subsequent characters can be letters, an underscore, dollar sign or digit.*
3. *White spaces are not allowed within identifiers.*
4. *Identifiers are case sensitive so **VALUE** is a different identifier than **Value***

| Average | Height | A1 | Area_Circle |
|---------|--------|----|-----------|

Invalid Identifiers are as follow:

| 2types | Area-circle | Not/ok |
|--------|-------------|--------|

### Naming Convention for Identifiers

- **Class or Interface-** These begin with a capital letter. The first alphabet of every internal word is capitalized. Ex: class **Myclass;**
- **Variable or Method –**These start with lower case letters. The first alphabet of every internal word is capitalized. Ex:- int **totalPay;**
- **Constants-** These are in upper case. Underscore is used to separate the internal word. Ex:-final **double PI=3.14**;
- **Package –** These consist of all lower-case letters. Ex:- **import java.io.*;**

## Data Types

Java is strongly typed language. The safety and robustness of the Java language is in fact provided by its strict type. There are two reasons for this: First, every variable and expression must be defined using any one of the type. Second, the parameters to the method also should have some type and also verified for type compatibility. *Java language 8 primitive data types:*

**The primitive data types are**: char, byte, short, int, long, float, double, boolean. These are again grouped into 4 groups.

1. **Integer Group**: The integer group contains byte, short, int, long. These data types will need different sizes of the memory. These are assigned positive and negative values. The width and ranges of these values are as follow:

| Name | Width | Range |
|------|-------|-------|
| long | 64 | −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| int | 32 | −2,147,483,648 to 2,147,483,647 |
| short | 16 | −32,768 to 32,767 |
| byte | 8 | −128 to 127 |

**byte:**

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from −128 to 127.Variables of type **byte** are especially useful when you're working with a ***stream of data*** from a network or file. They are also useful when you're working with ***raw binary data*** that may not be directly compatible with Java's other built-in types. Byte variables are declared by use of the **byte** keyword.
For example, the following declares two **byte** variables called **b** and **c**:

byte b, c;   // *where b and c are identifiers*

**short:**

**short** is a signed 16-bit type. It has a range from −32,768 to 32,767. It is probably the least-used Java type. Here are some examples of **short** variable declarations:
short s;
short t;

**int:**

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from −2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. We can store byte and short values in an int.

Example

int x=12;

**long:**
**long** is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed.
Example
long x=123456;

## 2. Floating-Point Group

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. These are used with operations such as square root, cosine, and sine etc. There are two types of Floating-Point numbers: float and double. The float type represents single precision and double represents double precision. Their width and ranges are as follows:

| Name | Width in Bits | Approximate Range |
| --- | --- | --- |
| double | 64 | 4.9e–324 to 1.8e+308 |
| float | 32 | 1.4e–045 to 3.4e+038 |

**float:**

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision.

Example:
**float height, price;**

**double:**

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All the math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values.

Example:
**double area,pi;**
**Example program to calculate the area of a circle**

```
import java.io.*;
class Circle
{
public static void main(String args[])
    {
```

```
              double r,area,pi;
              r=12.3;
              pi=3.14;
              area=pi*r*r;
              System.out.println("The Area of the Circle is:"+area);
        }
}
```

### 3. Characters Group

In Java, the data type used to store characters is **char.** However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, Java uses Unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human languages.

Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

**Here is a program that demonstrates char variables:**

```
// Demonstrate char data type.
class CharDemo
{
        public static void main(String args[])
        {
                char ch1, ch2;
                ch1 = 88; // code for X
                ch2 = 'Y';
                System.out.print("ch1 and ch2: ");
                System.out.println(ch1 + " " + ch2);
        }
}
```

### 4. Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**.

**Here is a program that demonstrates the boolean type:**
```
// Demonstrate boolean values.
class BoolTest
 {
public static void main(String args[])
{
        boolean b;
        b = false;
        System.out.println("b is " + b);
```

```
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");
        b = false;
        if(b) System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

## Literals

A literal is a value that can be passed to a variable or constant in a program. Literals can be numeric, boolean, character, string notation or null. A constant value can be created using a literal representation of it. Here are some literals:

| Interger literal | Character literal | Floating point literal | byteliteral |
|------------------|-------------------|------------------------|-------------|
| int x=25; | char ch=88; | flaot f=12.34 | byte b=12; |

## Comments

In java we have three types of comments: single line comment, Multiple line comment, and document type comment.

Single line comment is represented with // (two forward slashes), Multiple comment lines represented with /*………….*/ (slash and star), and the document comment is represented with /**……….*/.

## Separators

In Java, there are a few characters that are used as separators. The most commonly usedseparator in Java is the semicolon. As you have seen, it is used to terminate statements.The separators are shown in the following table:

| Symbol | Name | Purpose |
|--------|------|---------|
| ( ) | Parentheses | Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types. |
| { } | Braces | Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes. |
| [ ] | Brackets | Used to declare array types. Also used when dereferencing array values. |
| ; | Semicolon | Terminates statements. |
| , | Comma | Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement. |
| . | Period | Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable. |

## The Java Keywords

There are **50 keywords** currently defined in the Java language (see Table bellow). These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as names for a variable, class, or method.

| | | | | |
|---|---|---|---|---|
| abstract | continue | for | new | switch |
| assert | default | goto | package | synchronized |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

## Operators

An operator performs an operation on one or more operands. Java provides a rich operator environment. An operator that performs an operation on one operand is called *unary operator.* An operator that performs an operation on two operands is called *binary operator*.

Most of its operators can be divided into the following *four groups: arithmetic, bitwise, relational, and logical*. Java also defines some additional operators that handle certain special situations. In Java under *Binary operator* we have Arithmetic, relational, Shift, bitwise and assignment operators. And under **Unary operators** we have ++, - - , !( Boolean not), ~(bitwise not) operators.

### i. Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

| Operator | Result |
|----------|--------|
| + | Addition |
| − | Subtraction (also unary minus) |
| * | Multiplication |
| / | Division |
| % | Modulus |
| ++ | Increment |
| += | Addition assignment |
| −= | Subtraction assignment |
| *= | Multiplication assignment |
| /= | Division assignment |
| %= | Modulus assignment |
| − − | Decrement |

The operands of the arithmetic operations are of numeric type. The Boolean operands are not allowed to perform arithmetic operations. The basic arithmetic operators are: addition, subtraction, multiplication, and division.

*Example program to perform all the arithmetic operations*
### Arith.java

```java
import java.io.*;
class Arith
{
    public static void main(String args[])
    {
        int a,b,c,d;
        a=5;
        b=6;
        //arithmetic addition
        c=a+b;
        System.out.println("The Sum is :"+c);
        //aritmetic subtraction
        d=a-b;
        System.out.println("The Subtractio is :"+d);
        //arithmetic division
        c=a/b;
        System.out.println("The Dision is :"+c);
        //arithmetic multiplication
        d=a*b;
        System.out.println("The multiplication is :"+d);
    }
}
```

**The Modulus Operator**

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following demonstrates the %

<div align="center">**Modulus.java**</div>

```
// Demonstrate the % operator.
class Modulus
{
public static void main(String args[])
    {
            int x = 42;
            double y = 42.25;
            System.out.println("x mod 10 = " + x % 10);
            System.out.println("y mod 10 = " + y % 10);
    }
}
```

**Arithmetic Compound Assignment Operators**

Java provides special operators that can be used to combine an arithmetic operation withan assignment. As you probably know, statements like the following are quite common in programming:
a = a + 4;

In Java, you can rewrite this statement as shown here:

a += 4;

This version uses the += *compound assignment operator*. Both statements perform the same action: they increase the value of **a** by 4.

Here is another example,
a = a % 2;

which can be expressed as
a %= 2;

**Increment and Decrement**

The ++ and the – – are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

x = x + 1;

can be rewritten like this by use of the increment operator:

x++;

Similarly, this statement:
x = x - 1;

is equivalent to
x--;

**Note:** If we write increment/decrement operator after the operand such expression is called post increment decrement expression, if written before operand such expression is called pre increment/decrement expression

**The following program demonstrates the increment and decrement operator.**

**IncDec.java**

```java
// Demonstrate ++ and --
class IncDec
{
public static void main(String args[])
    {
        int a = 1;
        int b = 2;
        int c;
        int d;
        c = ++b; //pre increment
        d = a--; //post decrement
        c++;  //post increment
        d--; //post decrement
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

### ii.    The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

| Operator | Result |
|---|---|
| ~ | Bitwise unary NOT |
| & | Bitwise AND |
| I | Bitwise OR |
| ^ | Bitwise exclusive OR |
| >> | Shift right |
| >>> | Shift right zero fill |
| << | Shift left |
| &= | Bitwise AND assignment |
| I= | Bitwise OR assignment |
| ^= | Bitwise exclusive OR assignment |
| >>= | Shift right assignment |
| >>>= | Shift right zero fill assignment |
| <<= | Shift left assignment |

These operators are again classified into two categories: **Logical operators, and Shift operators.**

## The Bitwise Logical Operators

The bitwise logical operators are **&**, |, **^**, and **~**. The following table shows the outcome of each operation. The bitwise operators are applied to each individual bit within each operand.

| A | B | A I B | A & B | A ^ B | ~A |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

**The Bitwise NOT**
Also called the *bitwise complement,* the unary NOT operator, **~**, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

00101010
becomes
11010101
after the NOT operator is applied.

**The Bitwise AND**
The AND operator, **&**, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

     00101010     42
  **&** 00001111   15

00001010 10

**The Bitwise OR**

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
  00101010    42
| 00001111    15
  ───────────
  00101111 47
```

**The Bitwise XOR**

The XOR operator, **^**, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the **^**.

```
  00101010 (42)
^ 00001111 (15)
  ───────────
  00100101 (37)
```

**Using the Bitwise Logical Operators**

The following program demonstrates the bitwise logical operators:

**BitLogic.java**

```java
// Demonstrate the bitwise logical operators.
class BitLogic
{
public static void main(String args[])
{

int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b) | (a & ~b);
int g = ~a & 0x0f;
System.out.println(" a|b = " +c);
System.out.println(" a&b = " +d);
System.out.println(" a^b = " +e);
System.out.println("~a&b|a&~b = " +f);
System.out.println(" ~a = " + g);
}
}
```

**Bitwise Shift Operators: (left shift and right shift)**

**The Left Shift**

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times.

---

It has this general form:

*value << num*

Here, **num** specifies the number of positions to left-shift the value in **value**. That is, the **<<**moves all of the bits in the specified value to the left by the number of bit positions specified by *num.*

**The Right Shift**

The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number oftimes. Its general form is shown here:

*value >> num*

Here, **num** specifies the number of positions to left-shift the value in **value**. That is, the >>moves all of the bits in the specified value to the right by the number of bit positions specified by *num.*

### ShiftBits.java

```
class ShiftBits
{
    public static void main(String args[])
    {
        byte b=6;
        int c,d;
        //left shift
        c=b<<2;
        //right shift
        d=b>>3;
        System.out.println("The left shift result is :"+c);
        System.out.println("The right shift result is :"+d);
    }
}
```

### iii. Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

| Operator | Result |
|----------|--------|
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

### iv.    Short-Circuit Logical Operators (|| and &&)

Java provides two interesting ***Boolean operators*** not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as ***short-circuit*** logical operators.

When we use || operator if left hand side expression is true, then the result will be true, no matter what is the result of right hand side expression. In the case of && if the left hand side expression results true, then only the right hand side expression is evaluated.

Example 1:  (expr1 || expr2)          Example2: (expr1 && expr2)

**The Assignment Operator**

The *assignment operator* is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

*var = expression*;

Here, the type of *var* must be compatible with the type of *expression.* The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

**The ? Operator**

Java includes a special ***ternary* (three-way)** *operator* that can replace certain types of if-then-else statements. This operator is the **?**. It can seem somewhat confusing at first, but the **?** can be used very effectively once mastered. The **?** has this general form:

***expression1 ? expression2 : expression3***

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?** operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same type, which can't be **void**. Here is an example of the way that the **?** is employed:

**Test.java**

```
class Test
{
public static void main(String args[])
{
        int x=4,y=6;
        int res= (x>y)?x:y;
        System.out.println("The result is :"+res);
}
}
```

## Expressions

An expression is a combination of operators and/or operands. Expressions are used to create Objects, Arrays, and Assigning values to variables and so on. The expression may contain identifiers, literals, separators, and operators.

***Example:-***
        int m=2,n=3,o=4;
        int y=m*n*o;

## Operator Precedence Rules and Associativity

The precedence rules are used to determine the priority in the case there are two operators with different precedence. The Associativity rules are used to determine the order of evaluation, in case two operators are having the same precedence. Associativity is two types: Left to Right and Right to Left.

Table shows the order of precedence for Java operators, from highest to lowest. Notice that the first row shows items that you may not normally think of as operators: **parentheses, square brackets, and the dot operator**. Technically, these are called *separators*, but they act like operators in an expression. Parentheses are used to alter the precedence of an operation. As you know from the previous chapter, the square brackets provide array indexing. The dot operator is used to dereference objects.

| Highest | | | |
|---------|---|---|---|
| ( ) | [ ] | . | |
| ++ | – – | ~ | ! |
| * | / | % | |
| + | – | | |
| >> | >>> | << | |
| > | >= | < | <= |
| == | != | | |
| & | | | |
| ^ | | | |
| \| | | | |
| && | | | |
| \|\| | | | |
| ?: | | | |
| = | op= | | |
| Lowest | | | |

**Note:** The operators =,?:, ++, and – are having Right to Left Associativity. The remaining Operators are having Left to Right Associativity.

**Using Parentheses**

*Parentheses* raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:
a >> b + 3
This expression first adds 3 to **b** and then shifts **a** right by that result. That is, this expression can be rewritten using redundant parentheses like this:
a >> (b + 3)

However, if you want to first shift **a** right by **b** positions and then add 3 to that result, you will need to parenthesize the expression like this:
(a >> b) + 3

## Type Conversion and casting

There are two types of conversion. They are *Implicit Conversion, and Explicit Conversion*.

**Implicit Conversion**

In the case of the implicit conversion type conversion is automatically performed by java when the types are compatible. For example, the **int** can be assigned to **long**. The **byte** cane assigned to **short**. However, all the types are compatible, thus all the type conversions are implicitly allowed.For example, double is not compatible with byte.

Conditions for automatic conversion
1. The two types must be compatible
2. The destination type must be larger than the source type

When automatic type conversion takes place the *widening conversion* takes place. For example,

```
int a;    //needs 32 bits
byte b=45;  //needs the 8 bits
a=b;  // here 8 bits data is placed in 32 bit storage. Thus widening takes place.
```

**Explicit Conversion**

Fortunately, it is still possible obtain the conversion between the incompatible types. This is called explicit type conversion. Java provides a special keyword "**cast**" to facilitate explicit conversion. For example, sometimes we want to assign int to byte, this will not be performed automatically, because byte is **smaller** than int. This kind of conversion is sometimes called "*narrowing conversion*". Since, you are explicitly making the value narrow. The general form of the cast will be as follow:

```
destination_variable=(target type) value;
```

Here the target type specifies the destination type to which the value has to be converted.

Example
```
int a=1234;
byte b=(byte) a;
```

The above code converts the int to byte. If the integer value is larger than the byte, then it will be reduced to modulo byte's range.

**casttest.java**

```java
import java.io.*;
class casttest
{
        public static void main(String args[])
        {
                int a=258;
                byte b;
                b=(byte) a;

                System.out.print(" The result is :" +b);
        }
}
```

**output:**

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1.The 0.23 will have been truncated.

## Automatic Type Promotion in Expressions

The expression contains the three things: operator, operand and literals (constant). In an expression, sometimes the sub expression value exceeds the either operand.

For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**,or **char** operand to **int** when evaluating an expression. This means that the sub expression **a * b**is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression,**50 * 40**, is legal even though **a** and **b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid **byte** value, back into a **byte** variable.However, because the operands were automatically promoted to **int** when the expression

was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

byte b = 50;
b = (byte)(b * 2);

which yields the correct value of 100.

**The Type Promotion Rules**

Java defines several *type promotion* rules that apply to expressions. They are as follows:
- First,all **byte**, **short**, and **char** values are promoted to **int**, as just described.
- Then, if one operandis a **long**, the whole expression is promoted to **long**.
- If one operand is a **float,** the entireexpression is promoted to **float**.
- If any of the operands is **double**, the result is **double**.

## Flow of Control (Control Statements)

The control statements are used to control the flow of execution and branch based on the status of a program. The control statements in Java are categorized into 3 categories:

  i.   **Conditional Statement (Selection Statements/Decision Making Statements)**
  ii.  **Loops (Iteration Statements)**
  iii. **Branching Statements (Jump Statements)**
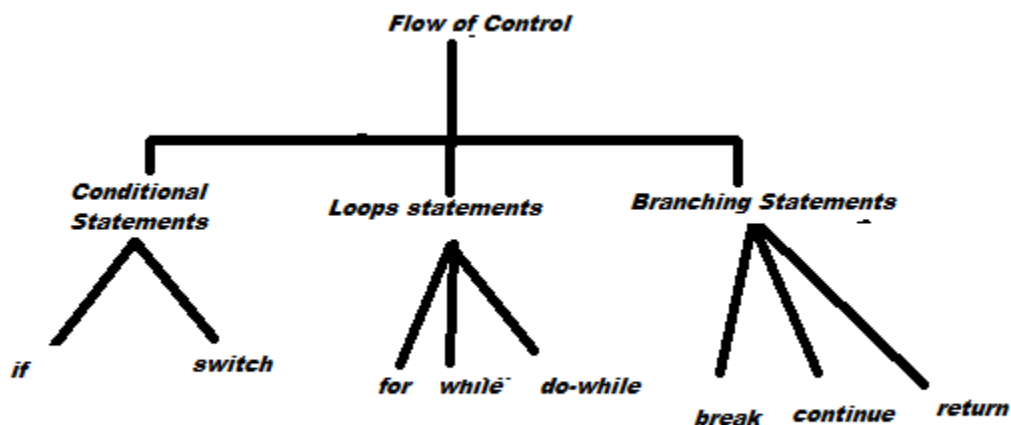
I.  **Conditional Statement (Selection Statements/Decision Making Statements)**: These include **if** and **switch**. These statements allow the program to choose different paths of execution based on the outcome of the conditional expression.

**if statement:** This is the Java's conditional branch statement. This is used to route the execution through two different paths. The general form of if statement will be as follow:
*if (conditional expression)*
*{*
     *statement1*
*}*
*else*
*{*
     *statement2*
*}*
Here the statements inside the block can be single statement or multiple statements. The conditional expression is any expression that returns the **Boolean** value. The else clause is optional. The if works as follows: if the conditional expression is true, then statement1 will be executed. Otherwise statement2 will be executed.

Example:
**Write a java program to find whether the given number is even or odd?**

**EvenOdd.java**
```
import java.io.*;
classs EvenOdd
{
        public static void main(String args[])
        {
                int n;
                System.out.println("Enter the value of n");
                DataInputStream dis=new DataInputStream(System.in);
                n=Integer.parseInt(dis.readLine());
                if(n%2==0)
                {
                        System.out.println(n+" is the Even Number");
                }
                else
                {
                        System.out.println(n+"is the ODD Number");
                }
        }
}
```

**Nested if:** The nested if statement is an if statement, that contains another if and else inside it. The nested if are very common in programming. When we nest ifs, the else always associated with the nearest if.

**The general form of the nested if will be as follow:**

```
if(conditional expresion1)
{
        if(conditional expression2)
        {
                statements1;
        }
        else
        {
                satement2;
        }
}
else
{
        statement3;
}
```

Example program:

 **Write a java Program to test whether a given number is positive or negative.**

<u>**Positive.java**</u>

```
import java.io.*;
class Positive
{
public static void main(String args[])  throws IOException
{
    int n;
    DataInputStream dis=new DataInputStream(System.in);
    n=Integer.parseInt(dis.readLine());
    if(n>-1)
        {
         if(n>0)
                System.out.println(n+ " is positive no");
        }
         else
                System.out.println(n+ " is Negative no");


}
}
```

**The if-else-if Ladder**

A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if ladder*. It looks like this:

**if**(*condition*)
      *statement***;**
**else if**(*condition*)
      *statement***;**
**else if**(*condition*)
      *statement***;**

**...**
**else**
      *statement***;**

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed.

Example Program:
**Write a Java Program to test whether a given character is Vowel or Consonant?**
<u>**Vowel.java**</u>

```
import java.io.*;
class Vowel
{
public static void main(String args[])  throws IOException
{
 char ch;
        ch=(char)System.in.read();
    if(ch=='a')
        System.out.println("Vowel");
    else if(ch=='e')
        System.out.println("Vowel");

    else if(ch=='i')
        System.out.println("Vowel");
    else if(ch=='o')
        System.out.println("Vowel");
    else if(ch=='u')
        System.out.println("Vowel");
    else

        System.out.println("consonant");



}
}
```

## The Switch statement

The **switch** statement is Java's multi-way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```
switch (expression)
{
        case value1:
                // statement sequence
                break;
        case value2:
                // statement sequence
                break;
...
        case valueN:
                // statement sequence
                break;
        default:
                // default statement sequence
}
```

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.

The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.

The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of "jumping out" of the **switch**.

**Write a Java Program to test whether a given character is Vowel or Consonant?( Using Switch)**

<div align="center">

**SwitchTest.java**

</div>

```java
import java.io.*;
class SwitchTest
{
public static void main(String args[])  throws IOException
{
```

```
    char ch;
    ch=(char)System.in.read();
    switch(ch)
    {
 //test for small letters
          case 'a': System.out.println("vowel");
               break;
       case 'e': System.out.println("vowel");
               break;
        case 'i': System.out.println("vowel");
               break;
        case 'o': System.out.println("vowel");
               break;
        case 'u': System.out.println("vowel");
               break;
       //test for capital letters
               case 'A': System.out.println("vowel");
               break;
        default: System.out.println("Consonant");
    }
}
}
```

The **break** statement is optional. If you omit the **break**, execution will continue on into thenext **case**. It is sometimes desirable to have multiple **case**s without **break** statements betweenthem. For example, consider the following program.

```
class Switch
 {
public static void main(String args[])
{
      int month = 4;
      String season;
      switch (month)
      {
            case 12:
            case 1:
            case 2:season = "Winter";
                    break;
            case 3:
            case 4:
            case 5:season = "Spring";
                    break;
            case 6:
            case 7:
```

```
                case 8:season = "Summer";
                        break;
                case 9:
                case 10:
                case 11:season = "Autumn";
                        break;
                default:season = "Bogus Month";
}
System.out.println("April is in the " + season + ".");
}
}
```

**Nested switch Statements**

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a*nested* **switch**. Since a **switch** statement defines its own block, no conflicts arise between the**case** constants in the inner **switch** and those in the outer **switch**. For example, the followingfragment is perfectly valid:

```
switch(expression)   //outer switch
{
        case 1: switch(expression)  // inner switch
                {
                        case 4: //statement sequence
                                break;
                        case 5://statement sequence
                                break;
                } //end of inner switch
                break;
        case 2: //statement sequence
                break;
        default: //statement sequence
} //end of outer switch
```

**There are three important features of the switch statement to note:**

- o  The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.

- o  No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.

- o  A**switch** statement is usually more efficient than a set of nested **if**s.

## II. Loops (Iteration Statements)

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

### i. while

The '*while*' loops is used to repeatedly execute a block of statements based on a condition. The condition is evaluated before the iteration starts. A *'for'* loop is useful, when we know the number of iterations to be executed in advance. If we want to execute the loop for indefinite number of times, a while loop may be better choice. For example, if you execute a *query to fetch data from database,* you will not know the exact number of records returned by the query.

**Syntax:**
**while(*condition*)**
**{**
        **// body of loop**
        **increment or decrement statement**
**}**

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

**Example program:**
**Write a java program to add all the number from 1 to 10.**
                                        **WhileTest.java**

```java
import java.io.*;
class WhileTest
{
    public static void main(String args[])
    {
        int i=1,sum=0;
        while(i<=10)
        {
         sum=sum+i;
        i++;
        }
        System.out.println("The sum is :"+sum);
    }
}
```

### ii. do-while statement

However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning.

Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is:

**do {**

      **// body of loop**

**} while (*condition*);**

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

**Example program:**
**Write a java program to add all the number from 1 to 10. (using do-while)**
<u>**WhileTest.java**</u>

```
import java.io.*;
class WhileTest
{
     public static void main(String args[])
     {
          int i=1,sum=0;
do
          {
           sum=sum+i;
           i++;
          }while(i<=10);
          System.out.println("The sum is :"+sum);
     }
}
```

**Note 1:** Here the final value of the i will be 11. Because the body is executed first, then the condition is verified at the end.

**Note 2:** The **do-while** loop is especially useful when you process a **menu** selection, because you will usually want the body of a menu loop to execute at least once.

**Example program: Write a Java Program to perform various operations like addition, subtraction, and multiplication based on the number entered by the user. And Also Display the Menu.**

**DoWhile.java**

```java
import java.io.*;
class DoWhile
{
    public static void main(String args[])  throws IOException
    {
        int n,sum=0,i=0;
        DataInputStream dis=new DataInputStream(System.in);

        do
        {
        System.out.println("Enter your choice");
        System.out.println("1 Addition");
        System.out.println("2 Subtraction");
        System.out.println("3 Multiplicaton");
        n=Integer.parseInt(dis.readLine());
        System.out.println("Enter two Numbers");
        int a=Integer.parseInt(dis.readLine());
        int b =Integer.parseInt(dis.readLine());
        int c;
        switch(n)
        {
            case 1: c=a+b;
                System.out.println("The addition is :"+c);
                break;
            case 2: c=a-b;
                System.out.println("The addition is :"+c);
                break;
            case 3: c=a*b;
                System.out.println("The addition is :"+c);
                break;
            default:System.out.println("Enter Correct Number");

        }

        } while(n<=3);

    }
}
```

###### iii.      for statement

The for loop groups the following three common parts into one statement: *Initialization, condition and Increment/ Decrement.*
**Syntax:**

> **for(*initialization*; *condition*; *iteration*)**

```
{
// body of the loop
}
```

The **for** loop operates as follows.
- ✓ When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once.
- ✓ Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
- ✓ Third, *Increment/Decrement* is used to increment /decrement the loop control variable value by one.

**Example program: same program using the for loop**

<div align="center">

**ForTest.java**

</div>

```
import java.io.*;
class ForTest
{
    public static void main(String args[])
    {
        int i,sum=0;
        for(i=1;i<=10;i++)
        {
      sum=sum+i;
 }
        System.out.println("The sum is :"+sum);
    }
}
```

**There are some important things about the for loop**
1. The initialization of the loop controlling variables can be done inside for loop.
   Example:
   **for(int i=1;i<=10;i++)**
2. We can write any Boolean expression in the place of the condition for second part of the loop.
   Example: where b is a Boolean data type
   boolean b=false;
   **for(int i=1; !b;i++)**
   **{**
   **        //body of the loop**
   **        b=true;**
   **}**
   This loop executes until the b is set to the true;

---

3. We can also run the loop infinitely, just by leaving all the three parts empty.
   Example:

   ```
   for( ;  ;)
   {
           //body of the loop
   }
   ```

**For each version of the for loop:**

A for loop also provides another version, which is called **Enhanced Version** of the for loop. The general form of the for loop will be as follow:

```
for(type itr_var:collection)
{
        //body of the loop
}
```

Here, type is the type of the iterative variable of that receives the elements from collection, one at a time, from beginning to the end. The collection is created sung the array.

**Example program:**
**Write a java program to add all the elements in an array?**
                                **ForEach.java**

```java
import java.io.*;
class ForEach
{
        public static void main(String args[])
    {
        int i, a[], sum=0;
             a=new int[10];
     a={12,13,14,15,16};
       for(int x:a)
        {
     sum=sum+x;
 }
        System.out.println("The sum is :"+sum);
    }
}
```

## III. Branching Statements (The Jump Statements)

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

    i.    **break statement**

---

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as "civilized" form of goto statement.

**Using break to Exit a Loop**

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

**// Using break to exit a loop.**

```
class BreakLoop
{
public static void main(String args[])
{
        for(int i=0; i<100; i++)
        {
                if(i == 10) break; // terminate loop if i is 10
                System.out.println("i: " + i);
        }
System.out.println("Loop complete.");
}
}
```

**Using break as a Form of Goto**

In addition to its uses with the **switch** statement and loops, the **break** statement can also be employed by itself to provide a "civilized" form of the goto statement. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can, for example, break out of one or more blocks of code.

The general form of the labeled **break** statement is shown here:

break *label*;

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of **break** executes, control is transferred out of the named block. The labeled block must enclose the **break** statement, but it does not need to be the immediately enclosing bloc.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followedby a colon. Once you have labeled a block, you can then use this label as the target of a**break** statement.

**Example code:**

```
class Break
{
public static void main(String args[])
{
boolean t = true;
first: {
second: {
third: {
        System.out.println("Before the break.");
        if(t) break second; // break out of second block
        System.out.println("This won't execute");
}
System.out.println("This won't execute");
}
System.out.println("This is after second block.");
}
}
}
```

**Running this program generates the following output:**

```
Before the break.
This is after second block.
```

### ii.      continue statement

It is used to stop the current iteration and go back to continue with next iteration. Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed. Here is an example program that uses **continue** to cause two numbers to be printed on each line:

**// Demonstrate continue.**

```
class Continue
{
public static void main(String args[])
{
for(int i=1; i<=10; i++)
{
```

```
            if (i%5 == 0) continue;
            System.out.print(i + " ,");}
}
}
```
Here all the numbers from 1 to 10 except 5 are printed. as  1,2,3,4,6,7,8,9,10.


### iii.    return statement

The last control statement is **return**. The **return** statement is used to explicitly return from
a method. That is, it causes program control to transfer back to the **caller of the method**.
As such, it is categorized as a jump statement.

Example code

```
class Test
{
        p s v main(String args[])      // caller of the method
        {
                int a=3,b=4;
                int x=method(a,b);//function call
                System.out.println("The sum is :"+x);
        }
        int method(int x,int y)      // called method
        {
                return (x+y);
        }
}
```
After computing the result the control is transferred to the caller method, that main in this case.


-----------------------**End of 1st Unit----------------**