

# Chapter 1 – Introduction to Java

**Marks Allotted: 16**

**Lectures: 10**

**Semester Pattern:**

<b>Exam:</b>	<b>Winter 2008</b>	<b>Summer 2009</b>
<b>Marks:</b>	<b>16</b>	<b>24</b>

**Yearly Pattern (JPR-IF 1526):**

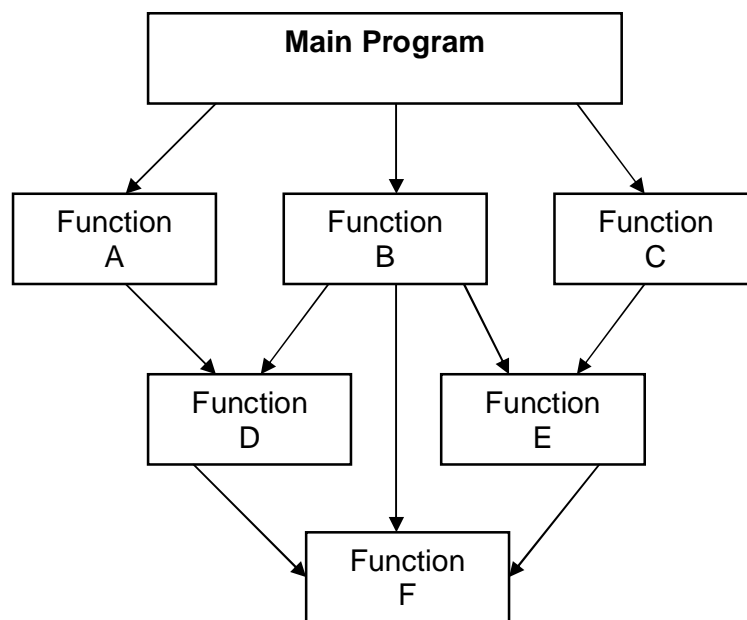
<b>Exam:</b>	<b>S'04</b>	<b>W'04</b>	<b>S'05</b>	<b>W'05</b>	<b>S'06</b>	<b>W'06</b>	<b>S'07</b>	<b>W'08</b>
<b>Marks:</b>	<b>16</b>	<b>16</b>	<b>20</b>	<b>16</b>	<b>16</b>	<b>20</b>	<b>12</b>	<b>12</b>

**Contents:**

- 1.1 Fundamentals of Object Oriented Programming  
Object and Classes, Data abstraction and encapsulation, Inheritance, Polymorphism, Dynamic Binding
- 1.2 Java Features  
Compiled and Interpreted, Platform independent and portable, Object oriented  
Distributed, Multithreaded and interactive, High performance
- 1.3 Constant, Variables and Data Types  
Constant, Data Types, Scope of variable, Symbolic Constant, Type casting, Standard default values
- 1.4 Operator and Expression  
Arithmetic Operators, Relational Operators, Logical Operators, Assignment Operator Increment and Decrement Operator, Conditional Operator, Bit wise Operator, Special Operator
- 1.5 Decision making and Branching  
Decision making with if statement, Simple if statement, The if else statement, The else if ladder, The switch statement, The? : Operator
- 1.6 Decision making and Looping  
The While statement, The do statement, The for statement, Jumps in Loops, Labeled Loops

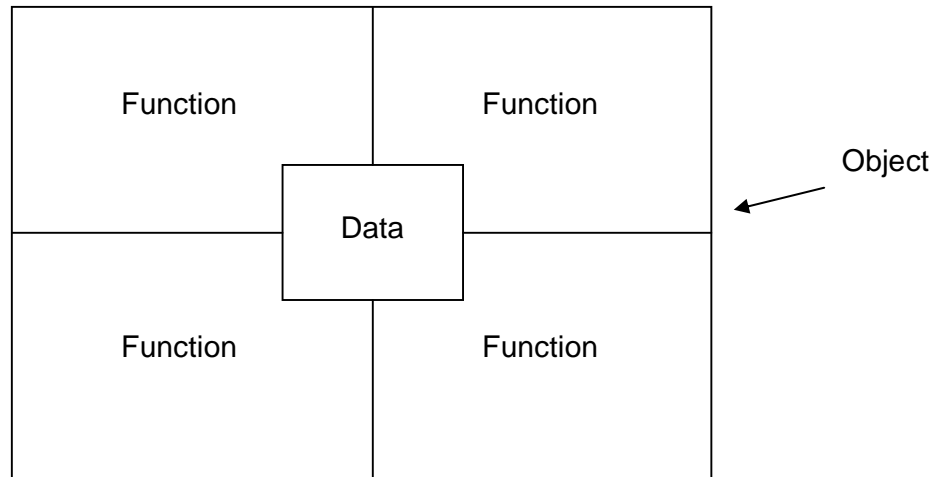
## Fundamentals of Object Oriented Programming

Our computer programs consists of two elements i.e. code and data. A program is conceptually organized around the data. According to the way of flow of the data, there are two types of programming paradigms. The first way is called procedure oriented programming. In which everything is organized in functions by functions. Flow of execution of the program is taken through the linear steps. The emphasis is given on the procedures rather than data. The data can be accessed anywhere in the functions. So there is no security for this data. C is the best example of such procedure oriented programming. C language employed this programming approach successfully.



**Fig 1.2 Procedure Oriented Methodology**

The second way is called object oriented programming. Basically, it is introduced to eliminate the flaws in the procedure oriented programming. It is the best way to manage the increasing complexity in procedure orientation. Object oriented programming organizes the program around the objects i.e. it treats the data as a critical element in the program development and does not allow it to flow freely around the system. It ties this data more closely to the functions that operate on it and protects it from unintentional modification by other functions. An object oriented programming can be characterized as 'data controlling access to code'. It has been given several organizational benefits.



**Fig 1.3 Object Oriented Methodology**

As shown in the above figure 1.3, the data is centered element in the object, which can be only be accessed by all the functions defined in it. (In Java's terminology, function is called as method).

Object Oriented Programming is having following main features.

- **Reduction in the complexity**  
Programs are divided into objects, which reduces the complexity of the program.
- **Importance of data**  
Emphasis is given on the data rather than procedure.
- **Creation of new data structure**  
Methods that operate on the data of an object are tied together in a single data structure.
- **Data Hiding**  
The data used in different object is hidden from outside world. They can not be accessed outside functions.
- **Characterization of the objects**  
Data structures are designed such a way that they characterize the objects.
- **Communication among objects**  
Objects in a single program can communicate with each other through methods.
- **Extensibility**  
New data and methods can be easily added whenever necessary. So, usability of the programs is extended.
- **Bottom-up programming approach**  
Object oriented programming follows Bottom-up programming approach in program design.

Object oriented programming design paradigm is most recently used programming approach. After C++ has implemented in a great extent, Java used is cleverly then C# and VB.net also implemented it afterwards.

## Basic Concepts in Object Oriented Programming

[Asked in W'06]

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. It is therefore necessary to understand some of the concepts used extensively in object oriented programming.

### Classes

The entire set of data and code of an object can be made using a user defined data type called as class. It is one of the types of abstract data types. In short, we can call the class is data type of object. Classes are user defined data types and they behave like the built in types of different programming languages. Once class has defined, we can create any number of objects of that.

### Objects

Object is called as the instance of the class. It is the run time entity in an object oriented system. In short, we can call object as a variable of type class. After creation, objects take up space in computer memory and have associated address like 'record' in Pascal and 'structure' in C. When the program is executed, object can establish the path of communication among them by passing messages in between. Take an example of an interactive game of NFS (Need For Speed). It consists of a set of racing cars. In this, car can be called as one class and we have to create different objects of that class. In order to create this we have to change the data related to that class i.e. dimension, color, maximum speed etc. Common attributes will remain constant for each object.

Each object is associated with data of type class with which they are created. A class is thus collection of objects of similar types. For example, Maharashtra, Gujarat, Bengal, Goa, Tamilnadu are the members of class India.

In object oriented programming languages, objects are represented with data and methods (functions) as shown in following figure.

Object Name	
Variable Name-1	<i>Value-1</i>
Variable Name-n	<i>Value-n</i>
Method Name-1 ( )	
Method Name-n ( )	

**Fig. 1.4 a) Representation of an object**

Batsman	
Name	
Innings	
average( )	
strike_rate( )	

**Fig. 1.4 b) Example of representation of an object**

See the above example, the class 'Batsman' has been declared. It contains data members Name and Innings as well as methods average ( ) and strike\_rate( ). In order to create the object of type Batsman we have to use following notation.

```
Batsman opener;
```

Here 'opener' is name of the object. It will contain instance of class Batsman and the separate copy of Name, Innings, average ( ) and strike\_rate( ). If we want to give values to this object, following notation is to be used.

```
opener.Name = "Sachin";
opener.Innings = 310; etc.
```

and for calling methods,

```
opener.average ( );
opener.strike_rate ( );
```

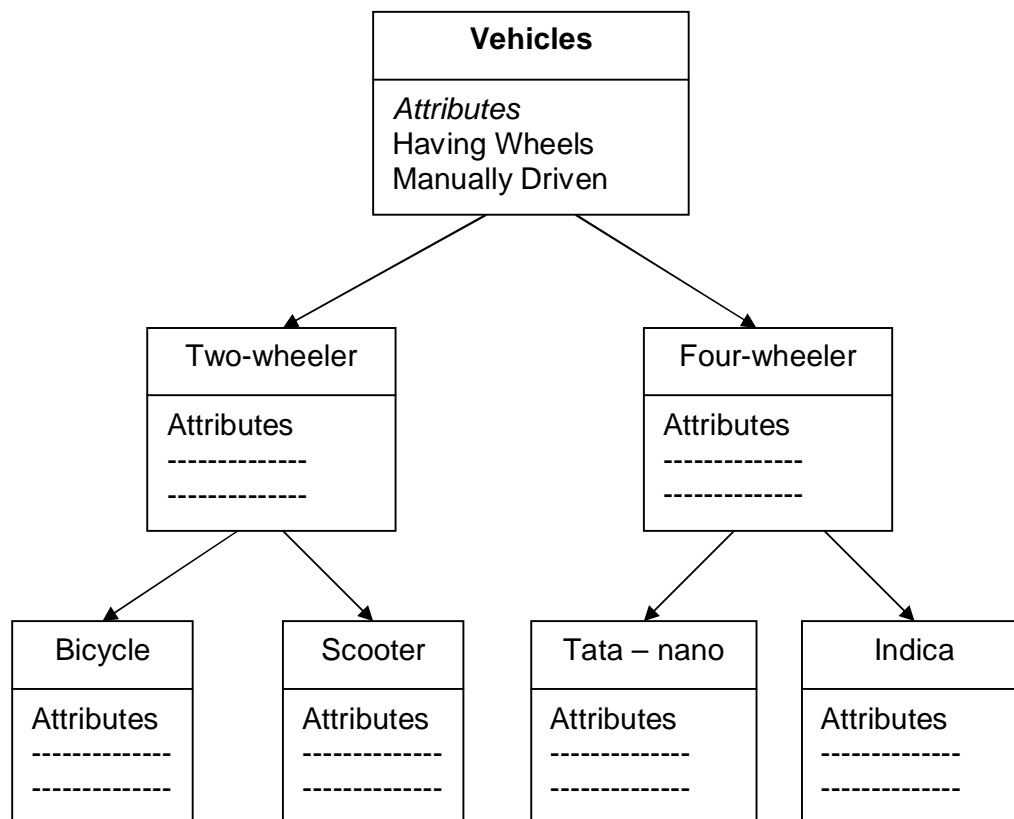
### **Data Abstraction and encapsulation**

The most striking feature of the class is 'Data encapsulation'. It refers to wrapping up of data and methods into the single unit. The data contained in the object is not accessible outside of the class. The methods which are wrapped around the object can access the data. These methods provide interface between object's data and program. This insulation of the data from direct access by the program is called 'data hiding'. Each object performs a specific task without any concern of the internal implementation. In C++, the data hiding is implemented by writing private variables and functions.

Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight, cost and methods that operate on these attributes. They encapsulate all essential properties of the objects that are to be created. Sometimes, the attributes are called data members because they hold information. Since classes use the concept of data abstraction, they are known as 'Abstract Data Type'.

## Inheritance

Inheritance is one of the most useful properties of object oriented programming. It is the process by which one class acquires the properties of objects of another class. It supports the concept of hierarchical classification. If we take the example of our family, son or daughter has properties of father or mother.



**Fig. Example of an inheritance**

See the example given in fig. 1.5. 'Vehicles' is the main class we can call it a root class, base class or super class. It is having its own attributes and methods. These properties are inherited to two different classes i.e. Two-wheeler and Four-wheeler. So, Two-wheeler and Four-wheeler both have wheels and can be manually driven. Furthermore, Two-wheeler and Four-wheeler classes can have their own uncommon

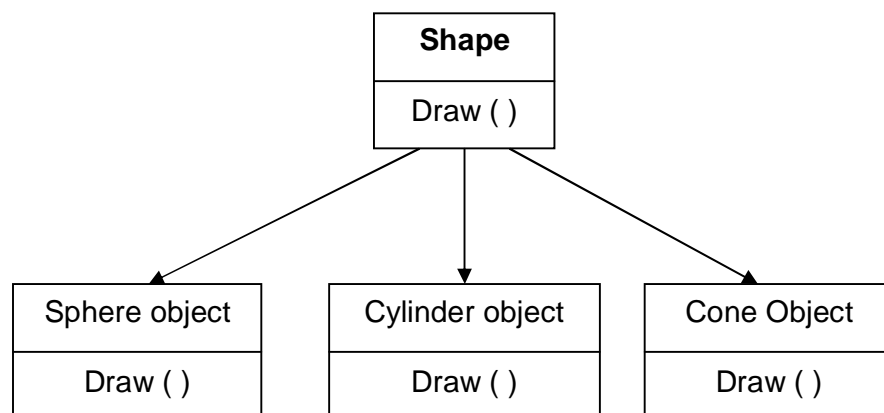
properties. These can also be inherited in their sub-classes i.e. Bicycle and Scooter for Two-Wheeler and Tata – nano and Lenova for Four-wheeler. Thus, there is no need to have the similar properties again in sub-classes. They can use the properties given in the base class or super class. We can conclude that the process of inheritance provides the idea of reusability. It is possible to add additional features to an existing class without modifying it. Thus, new class can also be created which will have combined features of both classes.

In inheritance, each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

## Polymorphism

Polymorphism is the Greek term means that ability to take more than one form. Same ability is having in 'object oriented programming' also. One name can be used for different functions. But, this function depends upon type of the data used in the actual operation. Take an example of one method called area ( ). When we pass one argument to this method, it will consider it as side of a square and according to that area of the square will be calculated. When we pass two arguments to area ( ), it will consider them as length and breadth of a rectangle and calculate area accordingly. And after passing three arguments to area ( ), these can be three sides of the triangle to calculate the area of it. So, here method area ( ) is having property of polymorphism i.e. ability to take more than one forms. Our method name is same but its data i.e. parameters decides its function.

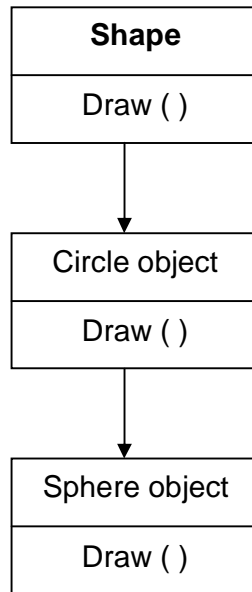
More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." Polymorphism plays an important role in objects having different internal structure to share their same external interface.



**Fig. Advantage of Inheritance and Polymorphism**

## Dynamic Binding

The term binding refers to assigning one thing to another. In programming languages, there are two types of bindings i.e. static and dynamic. Static binding means at the time of compilation and execution, compiler knows what to do and with which values? But, in case of dynamic binding compiler decides the function at run time i.e. at execution time. Thus, at the time of compilation, compiler does not know which method to call? This decision is made after execution of program. In C++ programming language, this concept is applied with 'virtual function'. This is also called as run time polymorphism.



**Fig. 1.7 Inheritance**

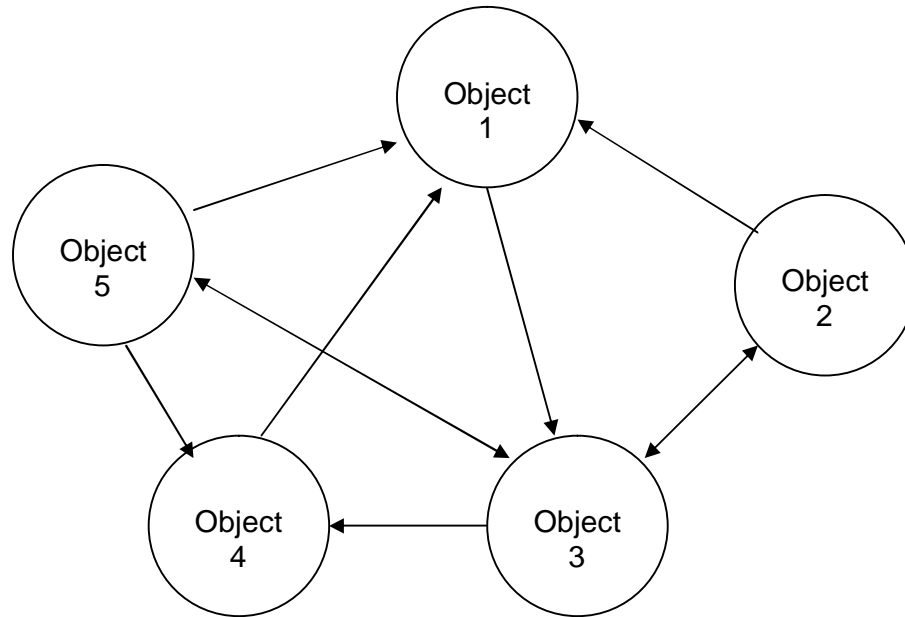
Consider the figure 1.7. The class Sphere is derived from Circle which is again derived from Shape. All the three classes contain Draw ( ) method. When we are calling, Draw ( ) method by using object of Sphere, which method will it call? It is decided using dynamic binding. By this, it is possible to call all three instances of Draw ( ) by creating objects of class Sphere. It is decided at run-time.

### Message Communication

An object oriented program consists of a set of objects that communicate with each other. This process of programming in an object oriented language involves following basic operations.

- Creating classes that define objects and their behavior.
- Creating objects from class definitions.
- Establishing communication among objects.





**Fig. 1.8 Communication among the objects**

Objects communicate with each other by sending and receiving information much the same way as people pass messages to one another as shown in the fig. 1.8. The object may contain the method to pass message from one object to another. This method is used along with name of the object.

### History of Java

Java is developed by group of Sun Microsystems engineers, led by an all-around computer wizard James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan. They wanted to design a small computer language that could be used for consumer devices like cable TV switchboxes on any platform. Since these devices do not have a lot of power or memory, the language had to be small and generate very tight code. They gave the code name to the project "Green."

As Sun scientist come from a UNIX background, they developed their language on C++ rather than Pascal. In particular, they made the language object oriented rather than procedure oriented. But, as Gosling said in the interview, "All along, the language was a tool, not the end." Gosling decided to call his language "Oak." (Because, he liked the look of an oak tree that was right outside of his window at Sun.) While obtaining the trademark for it, the scientists at Sun later realized that Oak was the name of an existing computer language, so they changed the name to Java. Many people believe that this name was suggested with abbreviation of **J**ames **G**osling, **A**rthur **V**an Hoff, **A**ndy **B**echtolsheim in 1995.

In 1992, the Green project delivered its first product, called “\*7.” It was an extremely intelligent remote control. Sun released the first version of Java in early 1996. It was followed by Java 1.02 a couple of months later. In 1998 Sun released the version of Java i.e. Java 1.2, which replaces the early toy-like GUI and graphics toolkits with sophisticated and scalable versions. Arthur Van Hoff rewrote the compiler of Java in Java itself which was originally wrote in C by James Gosling.

Sun promised that their language will have feature of “Write Once, Run Anywhere”™. In December 1998, the name was changed to Java 2. Since then, the core Java platform has stabilized. The current release, with the catchy name Java 2 Software Development Kit, Standard Edition version 1.6, is an incremental improvement over the initial Java 2 release, with a small number of new features, increased performance and, of course, quite a few bug fixes. Now that a stable foundation exists, innovation has shifted to advanced Java libraries such as the Java 2 Enterprise Edition and the Java 2 Micro Edition.

*Note: The history of ‘Java Programming Language’ given in Patrick Naughton’s book of ‘Java Handbook’ (see bibliography) in article ‘The Long Strange Trip to Java’.*



Java Logo



Duke, the Java mascot

**Fig. 1.9 we see Java in these.**

The Java project has seen many release versions. They are:

- JDK 1.1.4 (*Sparkler*) September 12, 1997
  - JDK 1.1.5 (*Pumpkin*) December 3, 1997
  - JDK 1.1.6 (*Abigail*) April 24, 1998
  - JDK 1.1.7 (*Brutus*) September 28, 1998
  - JDK 1.1.8 (*Chelsea*) April 8, 1999
- J2SE 1.2 (*Playground*) December 4, 1998
  - J2SE 1.2.1 (*none*) March 30, 1999
  - J2SE 1.2.2 (*Cricket*) July 8, 1999
- J2SE 1.3 (*Kestrel*) May 8, 2000

- J2SE 1.3.1 (*Ladybird*) May 17, 2001
- J2SE 1.4.0 (*Merlin*) February 13, 2002
  - J2SE 1.4.1 (*Hopper*) September 16, 2002
  - J2SE 1.4.2 (*Mantis*) June 26, 2003
- J2SE 5.0 (1.5.0) (*Tiger*) September 29, 2004
- Java SE 6 (1.6.0) (*Mustang*) December 11, 2006
- Java SE 7 (1.7.0) (*Dolphin*) anticipated for 2008

(J2SE – **Java 2 Standard Edition**)

## Features of Java

[\[Asked in W'08\]](#)

Java derives much of its character from C and C++. This is by intent. It has all the familiar syntax with C and object oriented features of C++. Because of the similarities between Java and C++, it is tempting to think of Java as simply the “Internet version of C++.” However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come

There were five primary goals in the creation of the Java language:

1. It should use the object-oriented programming methodology.
2. It should allow the same program to be executed on multiple operating systems.
3. It should contain built-in support for using computer networks (as Sun was especially working on Networking. Their slogan was ‘Network is the Computer’).
4. It should be designed to execute code from remote sources securely.
5. It should be easy to use by selecting what was considered the good parts of other object-oriented languages.

After actual development Java has achieved most of its features with enhancements. These are listed below.

1. Simplicity
2. Object Oriented
3. Platform Independence
4. Portable
5. Compiled and Interpreted
6. Robust
7. Secure
8. Distributed

9. Multithreaded
10. High performance
11. Architectural Neutral
12. Dynamic
13. Extensible

These features describe the potential of full Java language which has made Java the first application language of the World Wide Web. Java also becomes the premier language for general purpose stand-alone applications.

### **Simplicity**

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to become master in it. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier because Java is strictly object-oriented language than C++. If you are an experienced C++ programmer, moving to the Java will require very little effort. Because Java has the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java. Also, some of the more confusing concepts from C++ are either removed from Java or implemented in a cleaner, more approachable manner. Beyond its similarities with C/C++, Java has another attribute that makes it easy to learn: it makes an effort not to have surprising features. In Java, there are a small number of clearly defined ways to accomplish a given task.

### **Object Oriented**

[\[Asked in S'05\]](#)

Java is strictly or truly object-oriented language due to the following reasons.

1. Everything in Java is considered as the object including the source program because it is written in a class itself.
2. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance non objects.
3. Java implements all of the features of Object oriented programming including inheritance, polymorphism, dynamic binding etc. An extensive class library is also available in the core language packages.
4. The library of Java is created in terms of package. It is a collection of similar working classes. The methods are defined in the classes. In order to use these methods we need to import the package in our program.
5. The Java class library (API) itself implemented inheritance of classes in order to re-use the code. Thus simplifying the program also.

6. Most of the methods and constructors of the Java classes are overloaded. That is, Java itself implements the property of polymorphism in large extent.
7. Java does not support the multiple inheritance to avoid the duplication of data. But multiple inheritance is supported by the way of 'interface'. Using interface, we can create the multiple inheritance by avoiding duplication of data.
8. Java does not support the concept of global variables in any case. Thus data encapsulation is strictly supported.

### **Platform Independence**

[Asked in S'06]

Java has implemented 'Write Once Run Everywhere' strategy. Program written on one platform (operating system) can be run on any other operating system. Changes and up gradation in the operating system does not affect Java programs to run. This is the reason why Java has become popular language for programming on internet all over the world. We can download Java application or Applet from internet and execute it locally. Size of the data types used in Java programs is also machine-independent.

### **Portable**

Unlike C and C++, there are no "implementation-dependent" aspects of the specification in Java. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them. For example, an int in Java is always a 32-bit integer. In C/C++, int can mean a 16-bit integer, a 32-bit integer, or any other size that the compiler vendor likes. The only restriction is that the int type must have at least as many bytes as a short int and cannot have more bytes than a long int. Having a fixed size for number types eliminates a major porting headache. Binary data is stored and transmitted in a fixed format, eliminating the "big endian/little endian" confusion. Strings are saved in a standard Unicode format. The libraries that are a part of the system define portable interfaces. Java compiler generates bytecode instructions that can be implemented on any machine.

### **Compiled and Interpreted**

Java is the only language, which has compiler and interpreter both. This has been designed to ensure platform independence nature for the language. Due to this Java has been made a two-stage system. First, Java compiler translates the source code into bytecode instructions and there after in the second stage, Java interpreter generates machine code that can be directly executed by machine that is running Java program.

This provides portability to any machine for which a Java virtual machine has been written. It also allows for extensive code checking and improved security.

## **Robust**

Literal meaning of robust is 'healthy'. That is, Java puts a lot of emphasis on early checking for possible problems, later dynamic (run-time) checking, and eliminating situations that are error prone. Thus, Java is also called as Strictly Typed Language. The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.

This feature is also very useful. The Java compiler detects many problems that, in other languages, would show up only at run time. Language like Visual Basic or COBOL that doesn't explicitly use pointers, but in C, pointers are needed to access strings, arrays, objects, even files. In Visual Basic, we do not use pointers for any entities. On the other hand, there are many data structures that are difficult to implement in a pointer-less language. Java gives you the best of both worlds. We do not need pointers for everyday constructs like strings and arrays. We have the power of pointers if we need it. Exception handling mechanism of Java has made it more robust than others. Automatic garbage collection is also one of the features of this healthy language. That is, the objects that are no longer used in the programs are destroyed automatically by Java run-time system. While using the variables in the expressions we need to initialize them in the program. Though the main method of Java program is having compulsory command-line arguments, it is not necessary to give arguments always at the run-time.

## **Security**

Security becomes an important issue for a language that is used for programming on Internet. So, Java enables the construction of virus-free, tamper-free systems. Java systems not only verify all memory access but also ensure that no viruses are communicated with an application or an applet. The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization. Java has a far stronger security model than ActiveX since it controls the application as it runs and stops it from wreaking havoc. It also contains security manager class that determines what resources a class can access such as reading and writing to the local disk.

## **Distributed**

Java is designed for the distributed environment of the Internet/LAN, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra-address space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called

Remote Method Invocation (RMI). This feature brings an unparalleled level of abstraction to client/ server programming.

The networking capabilities of Java are both strong and easy to use. Anyone who has tried to do Internet programming using another language will find an easy process to open a socket connection in Java. An elegant mechanism, called servlets, makes server-side processing in Java extremely efficient. Many popular web servers support servlets.

### **Multithreaded Programming**

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. Java has a strong support of multithreaded programming. Implementation of multithreading in Java is simpler than any other languages that implements it. It makes the use of multithreading features of current platform or operating system. The Java runtime comes with the tools that support multi-process synchronization and construct smoothly running interactive systems. The benefit of multithreading is better interactive multimedia and real-time behavior.

### **High Performance**

Interpretation of bytecode slowed performance in early versions of Java, but advanced virtual machines with adaptive and just-in-time (JIT) compilation and other techniques now typically provide performance up to 50% to 100% the speed of C++ programs. This technology is being improved continuously and may eventually yield results that cannot be matched by traditional compilation systems. Java architecture is also designed to reduce overheads during runtime.

### **Architectural Neutral**

Many times operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. That is, any change made in the computer architecture does not affect the Java program to execute. The compiler generates an architecture-neutral object file format — the compiled code is executable on many processors, given the presence of the Java run time system.

### **Dynamic**

Being a dynamic language, Java is capable of dynamically linking in new class libraries, methods and objects. We can add the code to a Java application at run time. Java can also determine the type of class through



a query, making it possible to either dynamically link or abort the program, depending on the response.

## **Extensible**

Java program supports function written in other languages such as C/C++. These functions are known as native methods. This facility enables the programmer to use efficient functions available in other languages. Native methods are linked dynamically at run time.

## How Java is better than C++

Patrick Naughton has given the illustration on this issue in his book (*The Java Handbook*)

- **Global Variables**

Global variables in C++ program are the sign of a program that was not designed well enough to encapsulate the data in a sensible way. In Java, the only global name space is the class hierarchy. It is impossible to create global variables outside the classes.

- **Goto**

Powerful, well defined exception-handling embedded in Java removed the need for goto. This lack of an arbitrary branch statement allows the compiler to create more optimal code while still maintaining robustness and safety.

- **Pointers**

Improper arithmetic of pointers causes most of the bugs in today's code. It is trivial error to be "off by one" and smash an errant word past the end of the intended piece of memory. A "memory smash" is one of the most egregious of errors to detect and debug. Almost every computer virus ever written took advantage of a program's ability to probe and modify RAM using pointers. You can not convert an integer to pointer. You can not dereference an arbitrary memory address. Arrays are real defined objects, not merely addresses in memory. In Java you can not write past the end of an allocated array no matter hard you try.

- **Memory Allocation**

Memory Management in C and thus C++, is realized through the ying and yang of buggy code, the malloc() and free() library functions. Memory leaks with these functions causes a gradual slowdown of your program as it runs, while these unused garbage pages of memory are swapped to disk by the virtual memory system. Java has no malloc() and free() functions. Since, every complex data structure is an object, they are allocated via the new operator, which allocates space for an object on the heap of memory. You don't have to call free or delete. The garbage collector



runs whenever the system is idle, or when a requested allocation fails to find enough memory.

- **Fragile Data Types**

Liberal use of struct is C's poor excuse for a compound data type. Types like `int`, `char *` lead to impossibly non-portable code. Different C++ compilers will reserve 16, 32 or 64 bits for a given type depending on the natural size of the machine word.

- **Unsafe Type Casting**

Since objects in C++ are merely pointers to memory addresses, there is no way at runtime to detect if type cast is incompatible. Java object handles include complete information about the class that an object is an instance of, so it can do a runtime check for type compatibility, and issue an exception when it fails.

- **Unsafe Arguments List**

C++ has variable arguments facility (`varargs`) for the function. `Varargs` are a simple extension on the premise that any address can be mapped to any type leaving the task of checking type up to the programmer.

- **Separate Header Files**

There are no header files in Java. The type and visibility of class members is compiled into the Java class file. The Java interpreter enforces access control at runtime. So, there is no way to access the private variables outside the class.

- **Unsafe Structures**

C attempts to provide encapsulation of data via structure declaration called `struct` and polymorphism via a mechanism via `union`. These two constructs provide a thin veneer over critical and disastrous machine dependant alignment and size restrictions. Java does not have `struct` or `union` construct.

- **Preprocessor Hacks**

Java manages to do without a preprocessor, relaying on the `final` keyword to declare constants formerly created with `#define`.

## Java is popular on internet

[\[Asked in W'04'05\]](#)

The Java programming language is popular on internet due to the following reasons:

1. Java is meant to be used in distributed environments such as internet. Since both web and Java share the same philosophy, Java could be easily incorporated into the web system.
2. Java is a platform independent language. Its all applications can run as it on various operating systems.
3. Before Java World Wide Web was limited to display of still images only. However, incorporation of Java into web pages has made it capable of supporting animation graphics, games and wide ranges of special effects.

4. With the support of Java, web has become more interactive and dynamic. On the other hand, with the support of the web we can run Java programs someone else's computer across the internet.
5. Java is strongly associated with the internet because of the fact that the first application program written in Java was HotJava, a web browser to run applets on the internet.
6. Internet users can use Java to create applet programs and run them locally using a Java enabled browser such as HotJava or Java enabled browser to download an applet located on a computer anywhere on the internet and run it on his local computer. In fact Java applets have made the internet a true extension of storage of storage system of the local computer.
7. Internet users can set up their websites containing Java applet that can be used by other remote users of the internet. The ability of Java applet to ride on the information super highway has made Java a unique programming language.
8. We can create the Servlets in Java. Servlets are small programs that execute on the server side of a Web connection. Just as applets dynamically extend the functionality of a Web browser, servlets dynamically extend the functionality of a Web server. Servlets are generic extensions to Java-enabled servers. They are secure, portable, and easy to use replacement for CGI.

## The Java Language fundamentals

The best way to learn a programming language is to write some sample programs and execute them. Now, after discussion of object oriented programming concepts, let's look at our first program in Java i.e. printing "hello world" on the screen.

```
/* This is my first program in Java */
class MyProg
{
    public static void main(String args[ ])
    {
        System.out.println("Hello World");
    }
}
```

### **Program 1.1 Simple Java application.**

Have you noticed any similarities with a C or C++ program with this? It is having similarity with C and C++. So, let us discuss the program line by line.

#### **Comments**

The first line

```
/* This is my first program in Java */
```

This is called as a comment. It is written in same format that we uses in C and C++ application program. You can write anything in between `/*` and `*/`. This will not be compiled and executed by Java compiler. Java supports three different types of comments. We will see this afterwards.

## **Class Declaration**

The second line

```
class MyProg
```

As already mentioned, Java is strictly object oriented language. The whole program is also written in a class. Everything must be placed inside the class. Java does not allow anything written outside the class. Above statement declares a class in which the program can be written. `MyProg` is the name of the class which is a valid identifier name. Our program must also be saved with the same name i.e. `MyProg.java`. All Java applications and applets contain extension `.java`.

## **Opening curly brace**

Every class and method in Java is enclosed within the curly braces (`{` and `}`) same as in C++. Remember, number of opening braces must be equal to number of closing braces in a program. In class declaration, after closing brace of a class, there is no semicolon given. But, in C++ class definition, semicolon is in the syntax.

## **The main line**

```
public static void main(String args[ ])
```

This is the method where the actual execution of the program starts. Conceptually this is similar to the main function in C/C++. Every Java application program must include the `main( )` method. This is the starting point of the interpreter to begin the execution of program. A Java application can have any number of classes but only one of them includes `main( )` method to initiate the execution. Everything that is written in `main`'s curly braces is executed when program runs. The `main( )` method also contains the arguments called as command line arguments. Generally, given as string `args[ ]`, which contains an array of objects of type class `String`.

## The output Statement

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

This is the only executable statement inside the program and just similar to the `printf()` statement in C as well as `cout<<` construct in C++. As Java is strictly object oriented language, every user defined method or library method must be the part of an object of class. In above line, the `println` method is member of the `out` object, which is static data member of `System` class. This line will print the string

```
Hello World
```

on to the screen. The method `println` always appends a new line character ('\n') to the end of the string. Means, the subsequent output will start from the new line. If we want to continue with the same line then `print` method can also be used. Like, C and C++ every Java statement must end with a semicolon.

Lets look at another short program,

```
/* This is my second program */
class Second
{
    public static void main(String args[ ])
    {
        int num = 12, prog;
        prog = 5;
        num = num + (prog * 2);
        System.out.println("Value : ");
        System.out.print(num);
    }
}
```

### Program 1.2 a second short program

When you compile this program, you will get the output:

```
Value :
22
```

The declaration and expression statements written in `main()` method are just same as that in C/C++. Only output statements are different. By which Java is following the concept of strictly object oriented programming.

```
int num = 12, prog;
```

This is the declaration statement, which shows that variables `num` and `prog` are declared as integer. In these variables, we can store the

integer numbers. The variable num is declared with value, thus called as valued declaration. The value to the variable prog is given in next line. In the expression statement,

```
num = num + (prog * 2);
```

both the variables num and prog are having the values. Before using the variables in the expression the value must be given to them. If we eliminate the line,

```
prog = 5;
```

then the compiler will give following error:

```
Second.java:8: variable prog might not have been
initialized num = num + (prog * 2);
                        ^
```

This shows that, any variable that we are using in the expression must be initialized before actual use. The error has been shown by pointing a carrot (^) sign below the un-initialized variable. This has made Java more robust language.

## Java program implementation

Implementation of a Java program involves a series of three steps i.e.

9. Creating the program
10. Compiling the program
11. Running the program

Before we begin creating our program, the Java Development Kit (JDK) must be installed on our system. It can be freely downloaded from Sun Microsystems' website: <http://www.java.sun.com>

### Creating a program

We can create our program in any text editor such as notepad or dos editor. Specialized text editor for Java also available on the internet such as JCreator, JBuilder, Textpad, Eclipse, Gel etc.

Assume that the following program is created in one of the text editor.

```
//Test program
class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println("Hello...");
    }
}
```

```
        System.out.print("Welcome to the world of ");
        System.out.println("JAVA PROGRAMMING");
    }
}
```

We must save this file with the name HelloWorld.java. That is, the name of the Java application program is just same as name of the class which contains main() method. All the Java programs have the extension .java. This file is called as the source file. Remember that, if the program has multiple classes in it then name of the file will be the name of main method's class only.

### Compiling a program

Java Development Kit(JDK) has provided different tools for various activities. Among that javac is one of the tool called Java compiler which is used to compile the Java source file as,

```
javac HelloWorld.java
```

If there is no any error in the program, the javac creates a file named HelloWorld.class, which contains bytecode of the program. The compiler will automatically give the name to the class file.

### Running a program

As we have studied, that Java uses compiler and interpreter both. In the second stage of execution, we need to execute the program using Java interpreter. Java interpreter is a stand-alone program. Our program whose class file is created can be compiled like this,

```
java HelloWorld
```

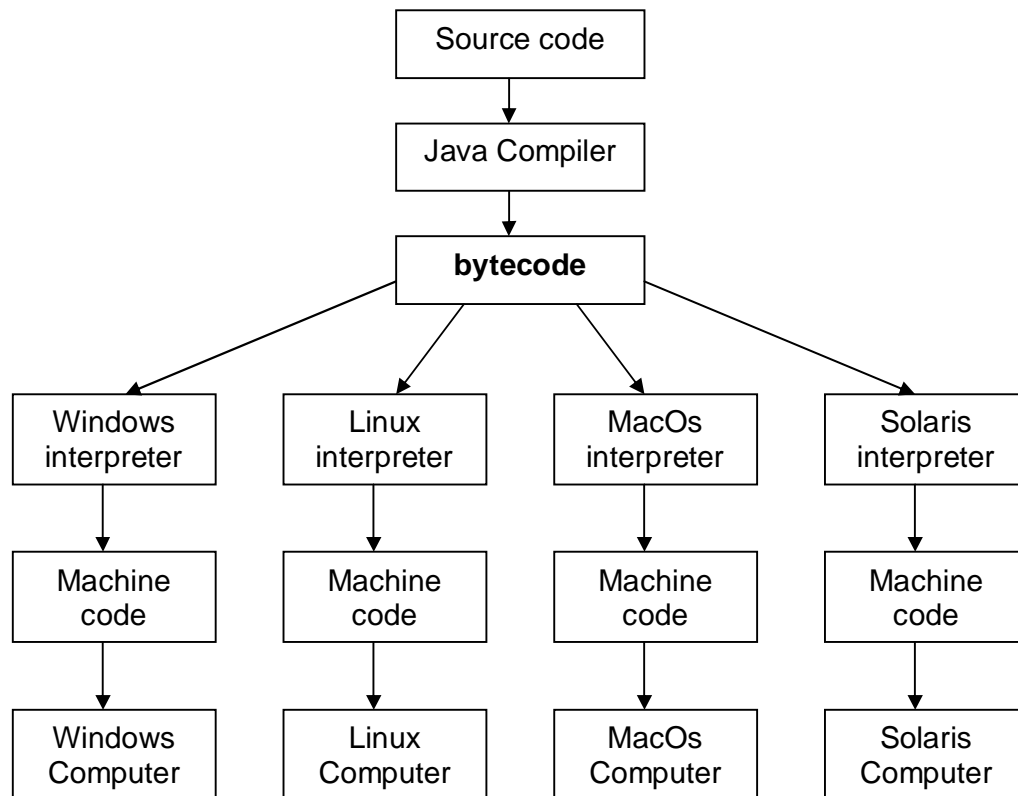
Now interpreter looks for the main method in the program and begins execution from there. When we execute the program, we will get following output.

```
Hello...
Welcome to the world of JAVA PROGRAMMING
```

### Use of bytecode

[\[Asked in S'04'07'09\]](#)

The Java compiler creates a bytecode. This is the machine independent code so; it can be executed on any machine. That is if you have created this bytecode on Windows operating system then it can be executed on Linux operating system.

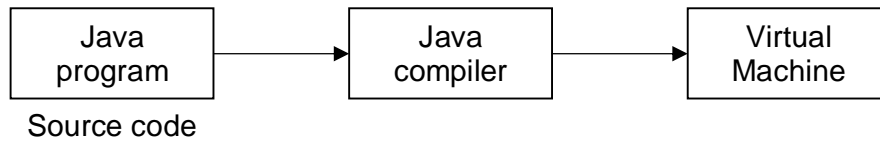


**Fig. 1.11 Creation of the machine independent code**

In order to make the Java as machine independent language, the bytecode is introduced. However, the Java Development Kit installed on different machine is different for all of them.

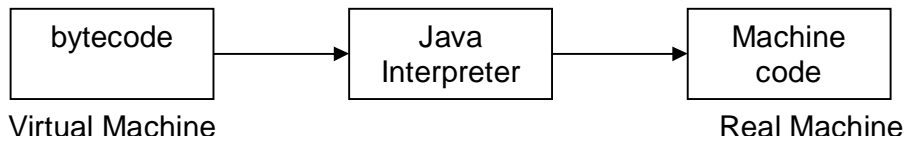
## Java Virtual Machine

All the programming language compilers translate the source code into machine code. Java compiler also performs the same function. However, the difference between Java and the other compilers is that Java compiler produces the intermediate code known as bytecode for the machine that does not exist. This machine is called as Java Virtual Machine. It exists only inside the computer memory. It is a simulated computer within the computer and does all the functions of the computer. Fig. 1.12 shows the process of compilation of a Java program into bytecode. This is also referred to as virtual machine code.



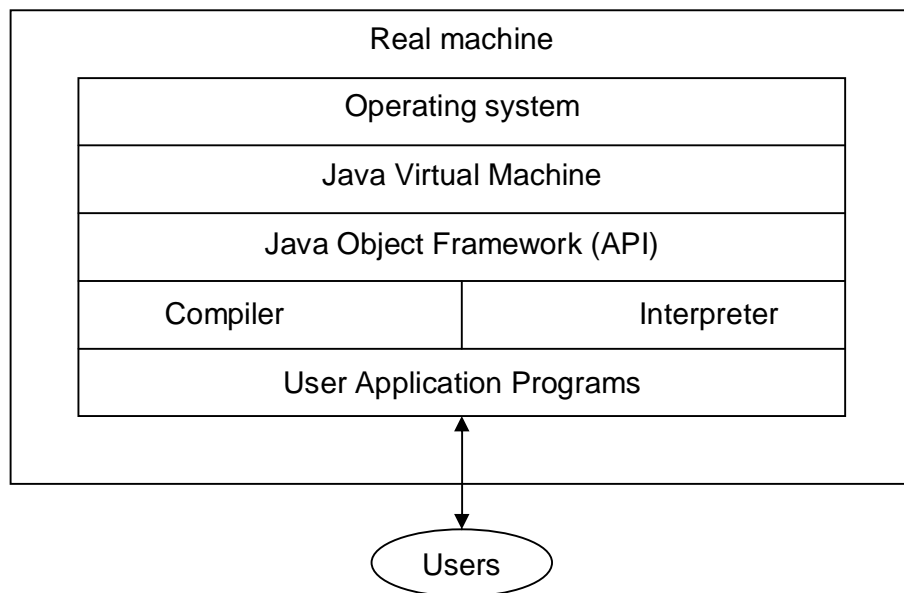
**Fig. 1.12 Process of compilation of Java program**

The virtual machine code is not machine specific. Java interpreter generates this code by acting as an intermediary between the virtual machine and real machine as shown in the fig. 1.13.



**Fig. 1.13 Process of converting bytecode into machine code**

Fig. 1.14 below illustrates how Java works on a typical computer. The Java object framework (Java API) acts as the intermediary between the user program and the virtual machine, which in turn acts as intermediary between the operating system and the Java object framework.



**Fig. 1.14 Layers of interaction of a Java Program**



## Java Program Structure

Documentation Section
Package Statements
Import Statements
Interface Statements
Class Definitions
<pre>main method class {     main method definition }</pre>

**Fig. 1.15 General Structure of a Java Program**

All programming language programs have their own structure of coding. Java is also having its own structure. A Java program may contain many classes of which only one class defines main method. Classes contain data members and methods that operate on the data members of the class. Methods may contain data type declaration and executable statements. To write a Java program, we first define classes and then put them together. The Java program may contain one or more sections as shown in figure 1.15.

### Documentation Section

This section consists of the set of comments giving name and descriptions of the program. It may contain anything that describes the program details but enclosed in comments. Comment can explain why and what classes and how related to algorithms. This helps to maintain 'program'. In addition to single line and multi-line comments discussed earlier Java add third style of comments (`/**.....*/`) called as documentation comments. This form of comments is used for generating the documentation automatically. Documentation section is not compulsory for all programs but suggested to use to increase the readability of program.

## Package Statement

The first statement allowed in a Java program is package statement. Java has organized library (in-built) classes in the form of packages. Thus, package can be called as a group of classes. If we want to declare our program as one of the part of the package, package statement is used. It informs the compiler that the classes defined in the program belong to this package.

Example:

```
package college;
```

After this statement, all the classes used in the Java program may become the part of user defined package, college. This statement is optional.

## Import Statement

This statement is similar to #include statement in C/C++. As Java has organized the classes in the form of packages, if we want to use any special class from Java's library we have to import that class or package using import statement.

Example:

```
import java.io.*;
```

This statement will instruct the compiler to import all the classes contained in the package java.io in our program in order to use them. Import can also be used to import the user defined packages.

## Interface Statements

Interface can be called as an 'Abstract Class' in Java. It contains group of method declarations. This is also optional statement and used only when we wish to implement the multiple inheritance feature in the program.

## Class Definitions

A Java program may contain multiple class definitions. Classes are the primary and essential elements of a Java program. All methods and variables that we declare in program must be the part of a class. By creating objects of it, we can call method with particular object. The number of classes used depends on the complexity and deepness of the problem.

## Main method class

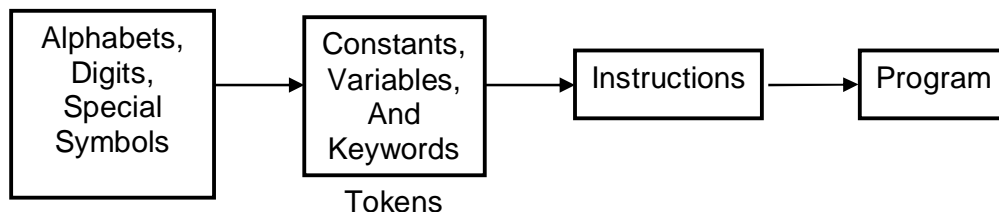
Programs in all programming languages are having the starting point. Since, every Java program requires a main method as the starting point. This class is the essential part of a Java program. A simple Java application program may contain only this part. The main method creates objects of various classes and establishes communication between them. Program execution starts in main and also ends in main.

## Java Tokens

A smallest individual element in a program is called as token. A Java application program is collection of classes. A class is defined by a set of declaration statements and methods containing executable statements. Most of the statements contain expressions, which describes the action carried out on data. The compiler recognizes the tokens for building up expressions and statements. In general, a Java program is collection of different tokens, comments, separators and white spaces. Java includes five different types of tokens. They are:

- Identifiers
- Keywords
- Literals
- Operators
- Separators

### The Java character set



**Fig. 1.16 Forming a Java Program**

Smallest units in Java language is characters which are used to write the tokens. These characters are defined by the Unicode character set, which is an emerging standard that tries to create characters for large number of scripts used worldwide.

The Unicode is the 16-bit character coding system and currently supports more than 34,000 defined characters derived from 24 languages across America, Europe, Gulf, Africa and Asia. However, generally we use only basic ASCII characters which include letters (upper and lower case), digits, punctuation marks and symbols used in mathematics. Therefore, we have used only ASCII character set (which is the subset of Unicode character set) in developing programs.

## Programming Related Issues

### Keywords

[Asked in S'05]

Keywords are the vital part of a programming language definition. These are also called as reserved words. Java has given a special meaning for certain words. These words are reserved for only the defined purpose in the programming. They can not be used for any other purpose. We can not use them for the name of variables, classes or methods. Understanding the meaning of all these words is most important for Java programmers.

All keywords are to be written only in lower case letters. Since, Java is case sensitive; we can use these words as name of variables, classes or methods by changing their case. However, being a good programmer it should be avoided.

Keywords give strength of the programming languages. Standard C compiler is having 32 keywords and a C++ compiler has 48 keywords whereas Java has defined 60 keywords. Although Java has modeled on C/C++, it does not use many of C/C++ keywords and on the other hand it has added as many as 27 additional keywords to implement new features of the language. All these Java keywords are listed in following table.

Abstract	boolean	break	byte	byvalue*
Case	cast*	catch	char	class
Const	continue	default	do	double
Else	extends	false	final	finally
Float	for	future*	generic*	goto*
if	implement	import	inner*	instanceof
Int	interface	long	native	new
Null	operator*	outer*	package	private
Protected	public	rest*	return	short
Static	super	switch	synchronized	this
Threadsafe*	throw	throws	transient	true
Try	var*	void	volatile	while

\* Reserved for future use.

**Table 1.2 List of Java keywords**

### Identifiers

Identifiers are the programmer-designed tokens. They are used for naming variables, classes, methods, objects, labels, packages and interfaces in a program. In order to give the name for the identifiers, we have to follow following rules:

1. They can have only alphabets, digits, underscore ( \_ ) and dollar sign ( \$ ) characters.
2. They must not begin with digit. Digit can be placed anywhere except starting position.

3. Uppercase and lowercase letters are different.
4. They can be of any length.
5. They must not be the keywords.
6. They can have any special symbol in it.

Identifier's name must be meaningful, short enough to be quickly and easily typed and long enough to be easily read. Java developers have followed some naming conventions rules which are also called as Hungarian notations. Java library has also implemented these notations. [\[Asked in S'04\]](#)

1. Name of all public methods and instance variables start with a lowercase letter.

Examples:

```
calculate
string
display
```

2. When the name contains more than one word, the second and subsequent words are marked with leading uppercase characters.

Examples:

```
findAverage
isLeapYear
grandTotal
```

3. All private and local variables use only lowercase letters combined with underscores.

Example:

```
birth_day
total
avg_marks
```

4. All class and interface names starts with an uppercase letter and each subsequent letter of the word with uppercase letter.

Examples:

```
HelloJava
FirstClass
Example1
```

5. Variables that represent constant values use all uppercase letters and underscores between words.

Examples:

```
MAX_MARKS
VALUE_OF_PI
GRADE
```

These are like symbolic constants in C/C++.

Generally these Hungarian notations are used to make the Java program as unique one. These are not considered as 'rules'.

## Literals

These are the sequence of characters (i.e. digits, letters and symbols) that represent constant values to be stored in variables. A

constant value in Java is created by using literal representation of it. Java specifies five major types of literals. They are:

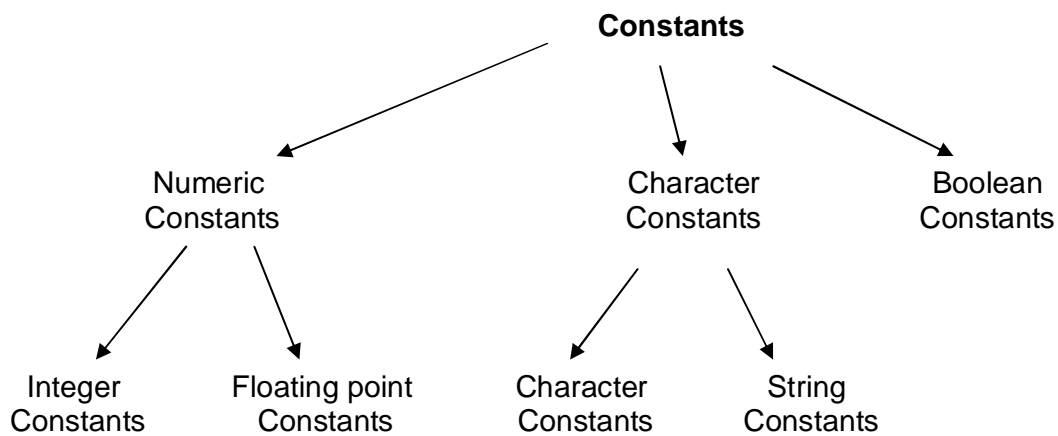
- i. Integer literals
- ii. Real literals
- iii. Character literals
- iv. String literals
- v. Boolean literals

Each of them has data type associated with it. This data type describes how the values behave and how they are stored.

## Constants

[Asked in S'05]

Constants are the values which do not change during the execution of program. Java supports several types of constants as shown in figure below.



**Fig. 1.17 Constants in Java**

### i. Integer Constants

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number. There are two other bases which can be used in integer literals, octal (base eight) and hexadecimal (base 16). Octal values are denoted in Java by a leading zero. Examples are 025, 041, 02 etc.

Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range. A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant with a leading zero-x, (0x or 0X). The range of a hexadecimal digit is 0 to 15, so A through F (or a through f ) are substituted for 10 through 15. Examples are 0x4A, 0x85, 0xF9 etc.

We rarely use the octal and hexadecimal integers in a program. Integer literals create an int value, which in Java is a 32-bit integer value.

## ii. Floating-point Constants

These are also called as real constants. Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. Standard notation consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.000, 53.19259, and 0.6667 represent valid standard-notation floating-point numbers. Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an E or e followed by a decimal number, which can be positive or negative. It is expressed in following format:

mantissa e exponent

Where mantissa is either a real number expressed in decimal notation or integer. The exponent is an integer with an optional + or – sign. Examples: 6.0032E23, 314159E–05, and 2e+100.

Floating-point constants in Java default to double precision. To specify a float literal, you must append an F or f to the constant. You can also explicitly specify a double literal by appending a D or d. doing so is, of course, redundant. The default double type consumes 64 bits of storage, while the less-accurate float type requires only 32 bits.

## iii. Character Constants

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as 'x', 'k', and '@'. For characters that are impossible to enter directly, there are several escape sequences, which allow you to enter the character you need, such as '\"' for the single-quote character itself, and '\n' for the new-line character. Following are the escape sequences used in Java.

Escape	Meaning
\n	New Line
\t	Tab
\b	Backspace
\r	Carriage Return
\\	Form feed

\'	Single quote
\"	Double quote

**Table 1.3 Escape Characters**

There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation use the backslash followed by the three-digit number. For example, '\141' is the letter 'a'. For hexadecimal, you enter a backslash-u (\u), then exactly four hexadecimal digits. For example, '\u0900' to '\u97f' is used for Devanagari character display. '\ua432' is a Japanese Katakana character.

#### iv. String Constants

String constants in Java are specified as they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"This is first and \nThis is second"
 "\"I love Java Programming\""
"156...+856"
"India@hotmail.com"
```

The escape sequences and octal\hexadecimal notations that were defined for character constants work the same way inside of string constants. One important thing to note about Java strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there is in other languages. C/C++ implements the string as array of characters, but in Java it is not the case. Here, string is the object type.

#### v. Boolean Constants

There are only two logical values that a boolean value can have i.e. true and false. The values of true and false do not convert into any numerical representation. The true constant in Java does not equal 1, nor does the false constant equal 0. In Java, they can only be assigned to variables declared as boolean, or used in expressions with Boolean operators.

### Separators

These are the symbols which are used to indicate where groups of codes are divided and arranged. They basically used to define the shape and function of the program code. The most commonly used separator is semicolon. All these separators are listed in the table 1.4.



Separator	Name	Purpose
( )	Parenthesis	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.
{ }	Curly Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code for classes, methods, and local scopes.
[ ]	Square brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminating the statement
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a 'for' loop statement.
.	Dot/Period	Used to separate package names from sub-packages and classes. Also used to separate a variable or method from a reference variable.

**Table 1.4 Separators in Java****Variables**[\[Asked in S'05\]](#)

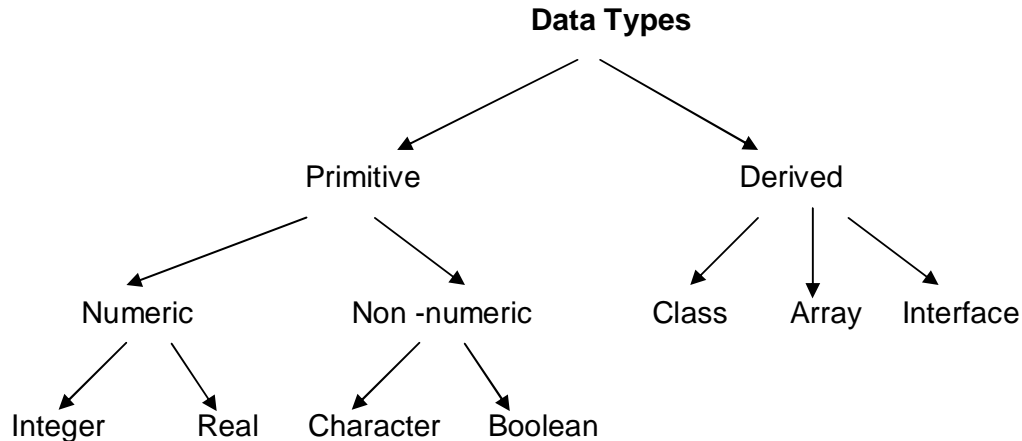
A variable is an identifier that denotes a storage location used to store the data value. Unlike constants that may remain unchanged during execution of the program, a variable may take different values at different times during execution of program. A variable name is chosen by the programmer in meaningful way so as to reflect what it represents in the program.

Example:

```
total_marks
calci
average
inventory etc.
```

Remember a variable must be a valid identifier name which will follow all the rules of its creation as studied earlier.

**Data Types**[\[Asked in S'05'07, W'05\]](#)



**Fig. 1.18 Data Types in Java**

Every variable declared in a Java program is having the data type. Data type specifies the size and type of values that a variable can have. Java has a rich set of data types. The variety of data types allows the programmer to select the type suitably as per the need of application. All these are categorized in two main types as shown in fig. 1.18.

Primitive types are also called as built in data types and derived data types are called as reference types.

### Integer Data Types

Integer type can hold the whole number such as 1822, -521 and 956 etc. Java supports four different type of integer data type as shown in the table 1.5.

Type	Size	Minimum Value	Maximum Value
byte	One byte	-128	127
short	Two bytes	-32,768	32,767
int	Four bytes	-2,147,483,648	2,147,483,647
long	Eight bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

**Table 1.5 Integer data types**

The integer data types supported by Java are byte, short, int and long. The maximum and minimum values that can be stored in these are shown in the table. The concept of unsigned data type is discarded in Java. According to the requirement of the programmer he/she can select the data type in programming.

Wider data types require more space and time for manipulation and it is advisable to use smaller data types, whenever possible. For example, we are storing a number 100 in integer data type will require more time and space for manipulation. It can also be stored in a byte data type. This will improve the speed of execution of program. We can make the integer long by appending the letter L at the end of the number. For example: 145L or 145l.

## Real Data Types

These are generally called as floating point data types. These are used to hold the numbers containing fractional parts such as 45.23 or 8966.120 etc. In short, real data types are used to store the floating point constants.

Type	Size	Minimum Value	Maximum Value
float	Four bytes	3.4e−38	3.4e+38
double	Eight bytes	1.7e−308	1.7e+308

**Table 1.6 Real data types**

Table 1.6 shows the real data type supported by Java. The float data type is called as single-precision real number and double is called as double-precision real number. They require four and eight bytes of memory respectively.

All the numbers written in a program with fractional point are treated as double precision numbers. In order to convert them into single precision, the letter f or F must be appended at the end of the number. Example:

45852.966f or 45852.966F

Double precision data types are needed when we need greater precision in storage for floating point numbers. All mathematical functions used in Math class returns double precision data type numbers. Floating point data types also support three special values called as NaN (Not-a-Number), Infinity (positive infinity) and - Infinity (Negative Infinity). All three are used to represent the result of operations when actual number is not produced. NaN used when 0.0 is divided by 0.0. Infinity is produced when a positive real number is divided by 0.0 and −Infinity is produced when a negative real number is divided by 0.0. These values can also be assigned to float and double variables using special constants in wrapper classes.

## Character Data Type

In order to store single character constant in a variable, character data type is used. Like C/C++, the keyword `char` is used to declare character data type variable. Two bytes are required to store a single character in memory. Because, the characters in Java use Unicode system, in which each character is represented by 16 bits i.e. 2 bytes. All characters in Unicode can be stored in a character data type variable.

## Boolean Data Type

This is one of the most useful data types of Java. Like bool data type of C++, Boolean data type is used to store two different values. But, values stored inside the boolean type will only be `true` and `false`. It can not have values like 0 or 1. It is used in program to test a particular condition during the execution. Boolean is denoted by keyword `boolean` and uses only one bit of storage. All relational operators (i.e. comparison operators like `<`, `>` etc.) return these boolean values.

## Declaring the variables

Variables are the names given to the storage locations. In order to use the variables in the program they must declared. The declaration does three things:

1. It tells the compiler what variable name is.
2. It specifies the data type of the variable.
3. Place of declaration in the program decides the scope of the variable.

A variable must be declared before used anywhere in the program. It can be used to store the value of any type studied earlier. General form of the declaration of the variable is:

```
data_type variable1, variable2, variable3.....variableN;
```

Variable names are separated by commas. It must be a valid identifier name.

Example:

```
int x;  
float avg, cal, s;  
char m, yes;  
boolean k, jack;
```

## Giving values to variables

A variable must be given a value after it has been declared but before it is used in the expression. This can be done in two ways.

1. By using assignment statement
2. By reading from the keyboard

### 1. By using assignment statement

The simple method of giving a value to the variable name is through the assignment statement as given below:

```
variablename = value;
```

For example:

```
Pi = 3.14f;
yes = 'y';
first = 150;
```

It is also possible to assign a value to a variable at the time of declaration. This takes the following form:

```
datatype variablename = value;
```

For example:

```
float Pi = 3.14f;
char yes = 'y';
short first = 150;
```

The process of giving initial values to variables is known as initialization. Remember, we must give a value to the variable (which is declared in a block) before it is used in the expression. But when we declare a variable as an instance variable (class variable), it is not necessary to give default values to it. It contains a default value. These default values of various data types are listed in following table 1.7.

Type of variable	Default value
boolean	false
byte	zero : 0
short	zero : 0
int	zero : 0
long	zero : 0L
float	0.0f
double	0.0d
char	null character

reference	null
-----------	------

**Table 1.7 Default values of various data types**

Consider the following code,

```
int x, y, z;
z = x * y;
System.out.println(z);
```

After compilation of this code, the compiler will flash following error.

```
variable x might not have been initialized
variable y might not have been initialized
```

So remember all the local variables declared in a Java program must be initialized before use.

## 2. By reading from the keyboard

We may also give values to the variables through by using various classes and their methods. One of the useful classes among them is **Scanner**. It is added by jdk1.5.

This class is defined in one of the library package of Java i.e. java.util. So in order to use it in our program we have to use following import statement at the top of our program.

```
import java.util.Scanner;
```

After this, the class Scanner will be available for our use in the program. We have to create the object of scanner in following way.

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

System.in is standard input handle i.e. keyboard. 'in' is the object of the Scanner class, which is then used to read any type of the data from keyboard in it. We have to use any of the following methods according to the requirement.

```
nextBoolean() for reading boolean values
nextByte()    for reading byte values
nextFloat()   for reading float values
nextDouble()  for reading double values
nextInt()     for reading int values
nextShort()   for reading short values
nextLong()    for reading long values
```

For example:

```
Scanner in = new Scanner(System.in);
int x;
System.out.println("Enter a number: ");
x = in.nextInt();
```

When the string "Enter the number: " is displayed onto the screen, the program control will hold until we enter anything from keyboard. After entering an integer value it will get stored in the variable, x.

Other useful classes are also there which are used for reading the data from keyboard such as, `DataInputStream`, `BufferedInputStream`, `BufferedReader` etc.

### Scope of variable

[Asked in S'04]

Basically, the Java variables are categorized into three types:

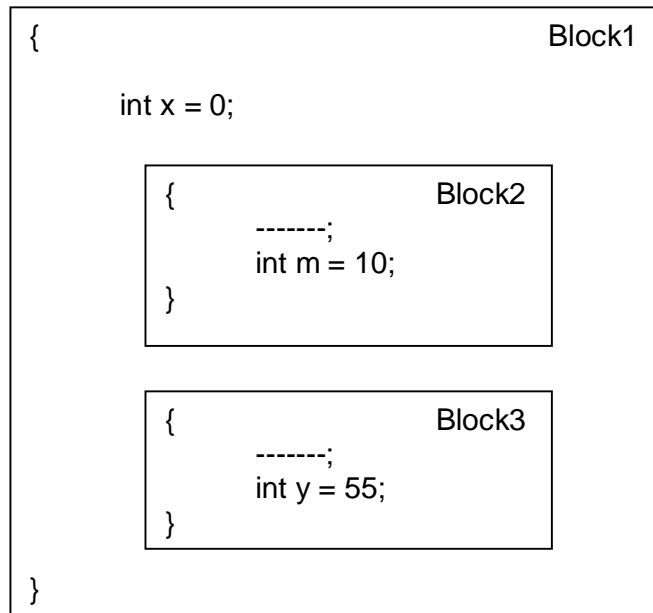
- Instance variables
- Local variables

The instance variables are declared inside the class. They are created when the objects of the class are created and therefore they are associated with the objects. They will have different values for different objects. It is not necessary to initialize all the instance variables; they will contain the default values as given below:

```
byte, int, and short: 0
long: 0L
double: 0.0d
float: 0.0f
char: null
boolean: false
reference: null
```

Variables declared inside a method or in a block are called as local variables. They are called so because they are not available for use outside of the particular block or method. They can be defined between an opening brace { and closing brace } in the program. These variables are visible to the program only from beginning of the brace to the end of closing brace. When the program control leaves the block their life ends.

Following figure illustrate the concept of local variables. Remember there is no concept of global variables in Java.



**Fig. 1.19 Block scope of variables**

In the figure 1.19, the variables *x*, *m* and *y* are local variables. Value of *x* is available in all three blocks. But the value of *m* is used only inside the block2 and value of *y* is available only in block3.

According to the types of variables, the scopes of the Java program have following types:

- Block scope
- Class scope

The instance variables are said to have class scope and local variables are said to have block scope.

## Operators

An operator is the symbol that takes one or more arguments and operates on them to produce a result. The constants, variables or expression on which operator operates are called as operands. Java supports a rich set of operators which are used in a program, to manipulate the data and variables. They are usually the part of mathematical or logical expression. Operators in Java are classified into number of categories:

1. Arithmetic Operators
2. Assignment Operators
3. Increment / Decrement Operators
4. Relational Operators
5. Logical Operators
6. Conditional Operators



## 7. Special Operators

### Arithmetic Operators

[Asked in W'08]

These include all the operators which are used to perform basic arithmetic operations. These are listed in table1.7.

Operator	Meaning	Example
+	Addition	9 + 45
-	Subtraction	89 - 12
*	Multiplication	10 * 3
/	Division	18 / 6
%	Modulo Division	14 % 3
+	Unary Plus	+51
-	Unary Minus	-92

**Table 1.7 Arithmetic Operators**

According to number of operands required for an operator, they are classified as Unary (one operand), binary (two operands) and ternary (three operands) operators. First five operators are binary operators as they require two operands. Last two operators are unary operators i.e. they require only one operand to operate. They are generally used to assign the sign for the constant.

When all the operands in an arithmetic expression are integer then it is called as integer expression. They always return integer value.

For example:  $x = 10$  and  $y = 3$

Then,  $x + y = 13$

$x - y = 7$

$x * y = 30$

$x / y = 3$  (decimal part is truncated)

$x \% y = 1$  (remainder of the division)

When an arithmetic expression involves only real operands is called as real arithmetic. A real operand may assume values either in decimal or exponent notation. Since floating point values are rounded to number of significant digits is permissible, the final value is approximation of the correct result.

When one of the operands is real and the other is integer, the operation is called as mixed-mode expression. If any one of the operands is floating point, then the operation will generate the floating point result.

Thus,

$25 / 10.0$  will generate result 2.5

Whereas,

25 / 10 will generate result 2

## Relational Operators

[Asked in S'07]

When we want to compare values of two variables for various means, relational operators are used. After comparing these values we may take several decisions in the program. Relational operators can be called as comparison operators. For example, we want to find the largest of two integers, we can use '>' operators to perform comparison. The expressions such as,

val > 12    or    cal < sal

contains the relational operator. So, they can be termed as relational expression. The value of relational expression can be either true or false. For example, if cal = 10 and sal = 15 then,

cal < sal    is true  
while  
sal > 20    is false

Java supports six different relational operators. These are listed and explained in the table 1.8 below.

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

**Table 1.8 Relational Operators**

Observe the following relational expressions which returns boolean values

36 > 42            true  
52.14 < 45        false  
10 < 8+9          false  
-74 != 0          false

When the arithmetic expressions are used on either side of a relational operator, the arithmetic expression will be evaluated first and then the results are compared. In short, the arithmetic operators are having highest priority over relational operators. Observe the following program and its output.

## Logical Operators

[Asked in S'07]

There are only three logical operators which are related with the logical decisions. These are listed in the table 1.9.

Operator	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

**Table 1.9 Logical Operators**

First two operators are used to combine the conditions i.e. to form a compound conditional expression. Literally, they have same meaning as in the digital techniques. Third operator is used to inverse the condition.

For example,

```
x > y && k != 12
```

This is termed as the logical expression or a compound relational expression. Whole expression also returns the boolean value. That is, the above expression will return true when the both the conditions ( $x > y$ ) and ( $k \neq 12$ ) are true. If any one of these conditions is false the expression will return false.

Consider another example,

```
m == 99 || j <= 10
```

This logical expression involves the logical OR operator. The expression will return true only when any one of the conditions ( $m == 99$  and  $j \leq 10$ ) is true. If both conditions are false, the whole expression will also return false. That is, in order to evaluate the expression to true, any one of the condition must be true.

The third operator is known as logical NOT operator. This is the only unary logical operator and generally used to negate the condition. If we write this operator in front of any other relational or logical expression, the result of the expression will be inverted. That is, true will become false and false will become true.

For example,

```
x = 36    then
!(x > 40)  will return true
!(x == 36) will return false
```

The following program will clear the idea of logical operators and expressions.

## Assignment operators

Assignment operators are used to assign the value of an expression to the variable. The general assignment operator is '='. This is used to assign the value given to its right side to the variable written on left side. Other assignment operators are also known as shorthand operators because they minimize our work of writing arithmetic expression. Assignment operators are listed in table 1.10.

Operator	Example	Equivalent Statement
+=	x += 10	x = x + 10
-=	x -= 10	x = x - 10
*=	x *= 10	x = x * 10
/=	x /= 10	x = x / 10
%=	x %= 10	x = x % 10

**Table 1.10 Assignment operators**

Following program will illustrate the use of assignment operators.

### Increment and Decrement operators

Like C and C++, Java is also having the increment and decrement operators' i.e.

++ and --

Both of these are unary operators. The operator ++ adds 1 to the operand and -- subtracts 1 from the operand. They are only associated with the variable name, not with the constant or the expression. They can be written in following from:

x++ or x--  
++x or --x

Both forms of the ++ increment the value of variable by one i.e. x++ or ++x will be equivalent to x = x + 1. As well as, x -- is equivalent to x = x - 1.

When the increment or decrement operator is used before variable, it is called as pre-increment or pre-decrement operator. And when it is used after variable, it is called as post-increment or post-decrement operator. The difference is simple. That is, when these pre-increment or pre-decrement operators are involved in the arithmetic expression, the values of respective variables will get affected before evaluation of expression. In case of post-increment and post-decrement operators, the value of variable will be affected after the evaluation of expression.

For example,

```
z = 14;
y = z++;
```

Here, the value of variable y will be 14 and z will be 15, because in the second expression, post increment operator is used. Value of variable z is assigned to y first and then it is incremented. If we change the second expression to,

```
y = ++z;
```

Now, both the values of y and z will be 15. Pre-increment operator does its job first then uses the value in the expression. Following program illustrates the use of increment and decrement operators.

### Conditional Operator

The only ternary operator (operator having three operands) is defined in Java called as conditional operator. The character pair ? : is termed as conditional operator. This is used to construct the conditional expression of the following form:

```
condition ? expression2 : expression3
```

When the condition written is true then the expression1 is evaluated else expression2 is executed. Means, any one of the expression1 and 2 is executed in any case. It can also be called as if-then-else operator.

Programming languages such as C and C++ are also having this operator. But, there is some difference between them. The conditional expression used in C++ and C, will not be assigned or used anywhere. But in Java, the independent conditional expression's return value must be assigned or used in the program.

For example,

```
x = 45;
y = 22;
x = (y>25) ? y : 50;
```

After completion of third statement execution, value of x will become 50. Because, the condition y>25 is false so first expression will not be executed. Only second expression is evaluated. That is, 50 will be assigned to variable x.

Following program illustrates the use of conditional operator.

```
// Conditional operator.
class Conditional
{
    public static void main(String args[])
```

```

    {
        int x = 25, y = 22;
        System.out.println("x = "+x);
        System.out.println("y = "+y);

        x = (y>25) ? y : 50;

        System.out.println("x = (y>25) ? y : 50");

        System.out.println("Now, x = "+x);
    }
}

```

### Program: Conditional operator.

Output:

```

x = 25
y = 22
x = (y>25) ? y : 50
Now, x = 50

```

Now, replace the line

```
x = (y>25) ? y : 50;
```

With

```
y>25 ? x = 22 : x = 25;
```

The unexpected type error will occur. This will only work in C/C++ but not in Java.

### Bitwise Operators

[Asked in W'04'08]

In order to manipulate the data at the bit level, the bitwise operators are provided in Java. These operators are used for testing the bits as well as shifting them to left or right etc. These can be applied to integer types only. That is, they can not be used along with float or double value. Table 1.11 shows the bitwise operators and their meaning.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise EX-OR
~	One's complement
<<	Left shift
>>	Right shift
>>>	Right shift with zero fill

### Table 1.11 Bitwise operators

All these operators except one's complement operator are binary operators. These perform the given bitwise operation on the operands. Accordingly, the result is obtained. Shift operators shift the binary equivalent bits of the operand to respective left or right position. >>> operator is one of the special type of bitwise operator which is generally used for the negative values which will be shifted to right by filling the zeroes to leftmost positions.

Following program explains the use of bitwise operators.

### Special Operators

Special operators are always unique to the programming language. Java supports two special operators. That is, instanceof and member selection operator or dot operator (.).

#### instanceof operator: [\[Asked in W'05\]](#)

instanceof is the keyword called as an object reference binary operator. It returns true if the object on the left-hand side is an instance or object of the class given on the right-hand side. This operator allows us to determine whether the object belong to particular class or not.

For example:

```
maharashtra instanceof India
```

Returns true if the object 'maharashtra' is the object of class 'India'. Otherwise it will return false.

#### Member selection operator

This is also called as dot operator (.). It is used to access the instance variable and methods of the class using objects.

For example:

```
company.salary()           //reference to method salary
company.employee           //reference to variable employee
```

This operator is also used to access the classes and sub-packages from packages.

### Arithmetic Expressions

An arithmetic expression is combination of variables, constants and operators as per the syntax of the particular programming language. In the previous programs we have used several different arithmetic and all other expressions. Java can handle any complex arithmetic expression. But, we have to take care of converting it into the format and appropriate syntax of the Java language.

Table 1.12 below gives some examples of arithmetic expressions.

Arithmetic Expression	Java expression
$(x+y)^2$	<code>(x + y)*(x + y)</code>
$x^2 + y^2 + 5$	<code>x * x + y * y + 5</code>
$(xy)(m-n)$	<code>x * y * (m - n)</code>
$\frac{ab}{c}$	<code>a * b / c</code>
$\frac{x}{y} + c$	<code>(x / y) + c</code>

**Table 1.12 Arithmetic Expressions**

These expressions are evaluated using an assignment statement of the form:

```
variable = expression;
```

Here, 'variable' is a valid Java variable name. When the statement is encountered, the expression is evaluated first. The result then replaces previous value of variable on the left-hand side. All the variables used in the expressions must be assigned values before evaluation is attempted.

For example:

```
a = (x + y)*(x + y);
b = (x / y) + c;
x = a * b / c;
```

### Precedence and Associativity of operators

When the expression involves more than one operation, then which operation is to be performed first is decided by the precedence of that operator. Highest precedence operation is solved first. Each operator of Java has precedence associated with it. The operators with the same precedence are evaluated according to their associativity. That is, the expression is evaluated either from left to right or right to left. The precedence and associativity of all these operators is given in the table 1.11.

Precedence	Operator	Description	Associativity
1	.	Member selection	L to R
	()	Function call	L to R
	[]	Array element reference	L to R
2	++, --	Postincrement, Postdecrement	R to L
3	++, --	Preincrement, Predecrement	R to L



	+, -	Unary plus, unary minus	R to L
	~	Bitwise compliment	R to L
	!	Boolean NOT	R to L
4	new (type)	Create object Type cast	R to L
5	*, /, %	Multiplication, division, remainder	L to R
6	+, - +	Addition, subtraction String concatenation	L to R
7	<<, >>, >>>	Shift operators	L to R
8	<, <=, >, >=	Less than, less than or equal to, greater than, greater than or equal to	L to R
	instanceof	Type comparison	L to R
9	==, !=	Value equality and inequality	L to R
	==, !=	Reference equality and inequality	L to R
10	&	Boolean AND	L to R
	&	Bitwise AND	L to R
11	^	Boolean XOR	L to R
	^	Bitwise XOR	L to R
12		Boolean OR	L to R
		Bitwise OR	L to R
13	&&	Conditional AND	L to R
14		Conditional OR	L to R
15	?:	Conditional Ternary	L to R
16	=, +=, -=, *=, / =, %=, &=, ^=,  =, Assignment <<=, >>=, >>>=		R to L

**Table 1.13 Precedence and Associativity of operators**

In case of arithmetic operators, as per the rules of arithmetic division or multiplication is performed first before addition or subtraction. In the same way, when the expression is compound expression, it is evaluated according to the precedence and associativity.

For example:

```
x = 10;
y = 22;
z = x++ * 10/y++ - x + y;
```

Here, value of z will be 16. As it is the compound statement it is solved according to priority as given below.

1. First increment operators will be evaluated i.e. x and y will become 11 and 23 respectively.

2. So the expression will look like,

$$z = 11 * 10/23 - 11 + 23$$

3. Now division and multiplication operators are having same precedence so they are evaluated from left to right, so perform multiplication first.

$$z = 110/23 - 11 + 23$$

4. Perform division now,

$$z = 4 - 11 + 23 \quad // \text{ division is performed in two integer numbers.}$$

5. Again addition and subtraction have same precedence so perform according to association i.e. from left to right. Subtraction is performed first.

$$z = -7 + 23$$

6. In this expression unary minus operator have highest precedence so 7 is considered as negative number and then addition is performed.

$$z = 16$$

This is our final answer.

### Type conversion

[Asked in S'05, W'06]

Java permits mixing of constants and variables of different types in an expression. But during the evaluation it follows very strict rules of type conversion as we have already studied that Java is strictly types language.

If the operands are of different type, the lower type is automatically converted to the higher type and the result is higher type. For example, if expression contains int, float and byte operands, then the int and byte will automatically be converted to float which is the higher type. Result of operations will also be float type. Following table 1.14 gives automatic type conversion chart.

	<b>char</b>	<b>byte</b>	<b>short</b>	<b>int</b>	<b>long</b>	<b>float</b>	<b>double</b>
<b>char</b>	int	int	int	int	long	float	double
<b>byte</b>	int	int	int	int	long	float	double
<b>short</b>	int	int	int	int	long	float	double
<b>int</b>	int	int	int	int	long	float	double
<b>long</b>	long	long	long	long	long	float	double

<b>float</b>	float	float	float	float	float	float	double
<b>double</b>	double	double	double	double	double	double	double

**Table 1.14 Automatic type conversion chart**

When one type of the data is assigned to another type of variable, automatic type conversion takes place if following two conditions are met.

1. Two types are compatible.
2. Destination type is larger than source type.

When these two conditions are met, a widening conversion takes place. While automatic type conversion, following factors are to be considered.

1. Conversion of float to int causes truncation of fractional part.
2. double to float conversion causes rounding of digits.
3. long to int conversion causes dropping of the excess higher order bits.
4. Type conversion can not be applied to boolean data type.

### The type promotion rules

Java defines type promotion rules that apply to expressions. They are as follows:

1. All byte, short and char values are promoted to int.
2. If one operand in the expression is long then the whole expression will be promoted to long.
3. If one operand is float then the whole expression will be promoted to float.
4. If any of the operands is double, the result is double.

Following program will demonstrate the type promotion rules.

```
// Type promotion rules
class Types
{
    public static void main(String args[])
    {
        byte b = 12;
        short s = 866;
        char c = 'm';
        int i = 25397;
        float f = 145.21f;
        long l = 865423L;
        double d = 6562.24;
        double value = (b*s) + (i/c) + (l+d) * f;
        System.out.println("b = "+b);
        System.out.println("s = "+s);
        System.out.println("c = "+c);
        System.out.println("i = "+i);
    }
}
```

```

        System.out.println("f = "+f);
        System.out.println("l = "+l);
        System.out.println("d = "+d);
        System.out.println("b*s = "+(b*s));
        System.out.println("i/c = "+(i/c));
        System.out.println("l+d = "+(l+d));
        System.out.print(" (b*s)+(i/c)+(l+d)*f : ");
        System.out.print(value);
    }
}

```

### Program 1.10 Type Promotion Rules

Output:

```

b = 12
s = 866
c = m
i = 25397
f = 145.21
l = 865423
d = 6562.24
b*s = 10392
i/c = 233
l+d = 871985.24
(b*s)+(i/c)+(l+d)*f : 1.2663160755479309E8

```

In the first sub-expression  $b * s$ , the result of operation is promoted to int. In second sub-expression  $i / c$ , the result of operation is promoted to int and in the third sub-expression  $l + d$ , the result of operation is promoted to double. Final expression contains all data types. In all these, double is the highest so the result of operation is double.

### Type casting

Although the automatic type conversions are helpful, they will not fulfill all our needs. We often encountered the situation where there is need to store the value of one type into variable of another type. In such situations, it is necessary to cast or temporary convert the variable of one type into another type. It is performed by type casting.

In automatic type conversion always lower type is converted to higher type. If we want to convert higher type to lower type then type casting is necessary. For that purpose the type casting operator is used.

This type of conversion is called as narrowing the conversion. It takes the following form.

```
(target_type) variable_name;
```

Here, target\_type specifies the desired type to convert the specified value to. For example, if we want to convert byte to int or long it will take following form.

```
byte b;
int val = 63;
b = (byte) val;
```

After this both b and val contain value 63. If we write the statement,

```
b = val;
```

Then the following error will occur,

```
possible loss of precision
found   : int
required: byte
```

So, without type casting it is not possible to assign higher data value to lower one. Now, if value of variable 'val' is changed to greater than 128, which is out of range of byte, then it will be automatically adjusted within the range of byte. The value of b will be -93.

Following program will explain this concept clearly.

```
// Type casting
class Casting
{
    public static void main(String args[])
    {
        byte b;
        int val = 163;
        b = (byte) val;
        System.out.println(b);
    }
}
```

### Program 1.11 Concept of type casting

Output:

-93

Four integer type can be cast to any other type except boolean. Casting into a smaller type may result in the loss of data. Similarly, float and double can be cast to any other except boolean. Casting a floating point value to an integer will result in the loss of fractional part. Table 1.13 gives the casts that result in no loss of information.

From	To
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double

int	long, float, double
long	float, double
float	double

**Table 1.15 Casting that results in no loss of information**

Observe one more program which demonstrates the concept of type casting.

```
// Type casting
class TypeCasting
{
    public static void main(String args[])
    {
        byte b;
        int val = 263;
        double d = 9563.25;
        long l = 56322145;

        System.out.println("int val  = "+val);
        System.out.println("double d = "+d);
        System.out.println("long l   = "+l);

        System.out.println("\nint to byte ");
        b = (byte) val;
        System.out.println(val+" to "+b);

        System.out.println("\ndouble to int ");
        System.out.println(d+" to "+(int)d);

        System.out.println("\nlong to double ");
        System.out.println(l+" to "+(double)l);

        System.out.println("\nlong to short ");
        System.out.println(l+" to "+(short)l);

        System.out.println("\ndouble to byte ");
        System.out.println(d+" to "+(byte)d);
    }
}
```

**Program: More type casting**

Output:

```
int val  = 263
double d = 9563.25
long l   = 56322145

int to byte
263 to 7
```

```
double to int  
9563.25 to 9563
```

```
long to double  
56322145 to 5.6322145E7
```

```
long to short  
56322145 to 26721
```

```
double to byte  
9563.25 to 91
```

## Symbolic Constants

We have already known certain universal constants. For example, value of Pi ( $\pi$ ) is 3.14, value of gravitational constant is 9.81 etc. In the program also it is possible to define the constants. If you are familiar with C or C++ the constants are defined as,

```
#define Pi 3.145      or  
const float Pi = 3.145
```

Same way if we want to define the constant in Java one type modifier is used called **final**. 'final' is the keyword used to define the symbolic constant in a Java program. It takes the following form:

```
final data_type variable_name = value;
```

The value assigned to the variable name is constant and can not be changed in any case in the program. At the time of declaration, it is necessary to assign the value to the variable.

For example:

```
final double Pi = 3.145;  
final float gvt = 9.81f;  
final int maximum = 1000;
```

We can use the symbolic constants in program in any expression but can not modify their value.

## Programming Style

Java is a free-form language. That is, we do not have to indent any lines to make the program work properly. The Java system does not take care where on the line we begin typing. This may be the license of bad programming but can be one of the advantages also.

For example,

```
System.out.println("Welcome to Java");
```

can also be written as,

```
System.out.println  
("Welcome to Java");
```

Or even as,

```
System.  
out  
.  
println("Welcome  
to Java");
```

but, we try to avoid this.

## Decision making and branching

[\[Asked in S'09\]](#)

Control flow is the heart of any program. It is the ability to adjust (or to control) the way that program progresses. By adjusting the direction of flow programs can become dynamic. Without control flow, programs would not be able to do anything more than several sequential operations.

Java supports different control flow and branching statements given below.

1. **if** statement
2. **if-else** statement
3. **switch-case** statement
4. Conditional operator statement

### Decision making with an 'if' statement

The 'if' statement is one of the powerful decision-making statement used to control the flow of execution. The way of execution of 'if' is just like in any other language's 'if' statement. It is syntactically identical to 'if' statement in C, C++ and C#. It takes following form:

```
if (condition)  
    statement;
```

or

```
if (condition)  
{  
    statement1;  
    statement2;  
    statement3;
```

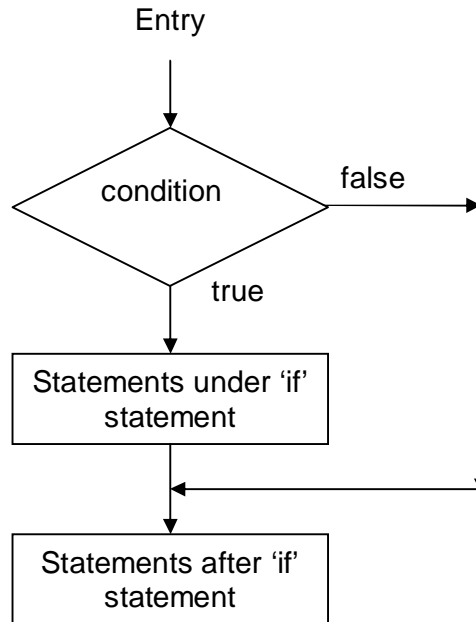


```

}

```

This allows the compiler to evaluate the condition first, which is the boolean expression. Then depending upon the value of the condition (true or false), the related statements are executed. That is, the execution of statement(s) depends upon the condition given in the 'if' statement. Diagrammatically, it can be shown as in fig. 1.20.



**Fig. 1.20 Control flow of an 'if' statement**

The condition generally involves relational and logical operators which return boolean value. Some examples of an 'if' statements are:

1. 

```
if(number < 0)
    System.out.println("The number is negative");
```
2. 

```
if(ch > 'A' && ch < 'Z')
    System.out.println("It is upper case letter");
```
3. 

```
if(sell_price > cost_price)
    System.out.println("You made profit");
```

All of the above statements involve the use of relational expression in the condition. Second example is having logical expression with logical AND operator. Instead of relational or logical expression, it is possible to write boolean variable also as shown in following example.

```

boolean correct = true;
if(correct)
    System.out.println("You are right...!");

```

The output of this statement is 'You are right...!' As the boolean variable 'correct' has value true it can be acted as the condition. Following program demonstrates the use of an 'if' statement.

### Nested 'if' statement

Nesting of the statements is one of the useful features of the all the programming languages. Nesting means we can write one 'if' statement in another 'if' statement. It takes the following forms:

1.    `if(condition1)`  
        `if(condition2)`  
        `statement;`
2.    `if(condition1)`  
        `if(condition2)`  
        `{`  
        `statement1;`  
        `statement2;`  
        `statement3;`  
        `}`
3.    `if(condition1)`  
        `{`  
        `if(condition2)`  
        `{`  
        `statement1;`  
        `statement2;`  
        `statement3;`  
        `}`  
        `statement4;`  
        `}`

In all the above cases, if both the conditions (condition1 and condition2) are true then only the respective statements are executed. It will just be equal to combining both conditions in a single 'if' statement using logical AND operator. The nesting of 'if' can be created in any depth.

Observe the following example to clear the idea of nesting of the 'if' statements.

```
// Nested ifs
class NestedIf
{
    public static void main(String args[])
    {
        int number = -52;
        System.out.println("Number is : "+number);
    }
}
```

```

        if(number > 0)                //1
            if(number%2 == 0)
                System.out.println("It is positive & even");

        if(number < 0)                //2
            if(number%2 == 0)
                System.out.println("It is negative & even");

        if(number > 0)                //3
            if(number%2 != 0)
                System.out.println("It is positive & odd");

        if(number < 0)                //4
            if(number%2 != 0)
                System.out.println("It is negative & odd");
    }
}

```

### Program 1.15 Nested 'if' statement

Output:

```

Number is : -52
It is negative & even

```

According to the value of variable 'number', the 'if' conditions are executed. In the first 'if', first condition is false so second will not be checked. In the second 'if', both conditions are true so we get the above output. In the third 'if', first condition is true but second is false. And in last one, first condition is false so second will automatically be ignored.

### The 'if-else' statement

The if-else statement defines both the actions which are to be taken on when condition is true and when condition is false. It is an extension of general if statement. The general form of if-else statement is:

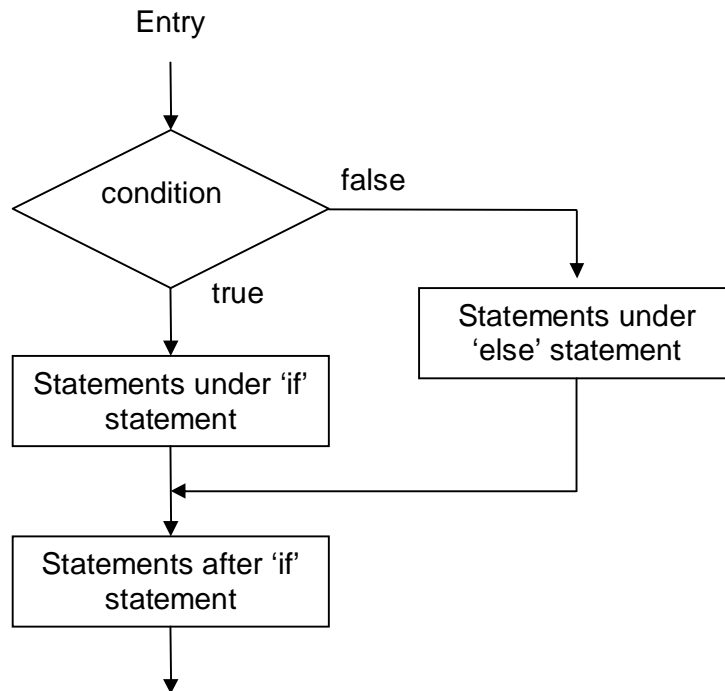
```

if(condition)
{
    statements for condition is true;
}
else
{
    statements for condition is false;
}
Statements after the blocks;

```

If the condition given is true then the statements inside the if-block are executed otherwise statement inside else-block are executed. That is, any one of the statements is executed depending upon the status of the condition (i.e. true or false).

After all the statements written after the blocks are executed though the condition is true or false. This concept is illustrated in fig. 1.21 with flowchart.



**Fig. Flowchart for if-else control structure**

'if-else' is one of the most useful control structures than that of independent if. It is applicable in many of the application programs. The condition given in the 'if' also combines relational and logical operators as well as boolean variables too.

### **Nested if-else structure**

'if-else' structure can also be nested but in such cases 'if's must be matched with respective else. It follows the following syntax:

```

if(condition1)      //first 'if'
{
    statement1;
    if(condition2) //second 'if'
    {
        statement2;
        statement3;
    }
    else             //second 'else'
    {
        statement4;
        statement5;
    }
}
  
```

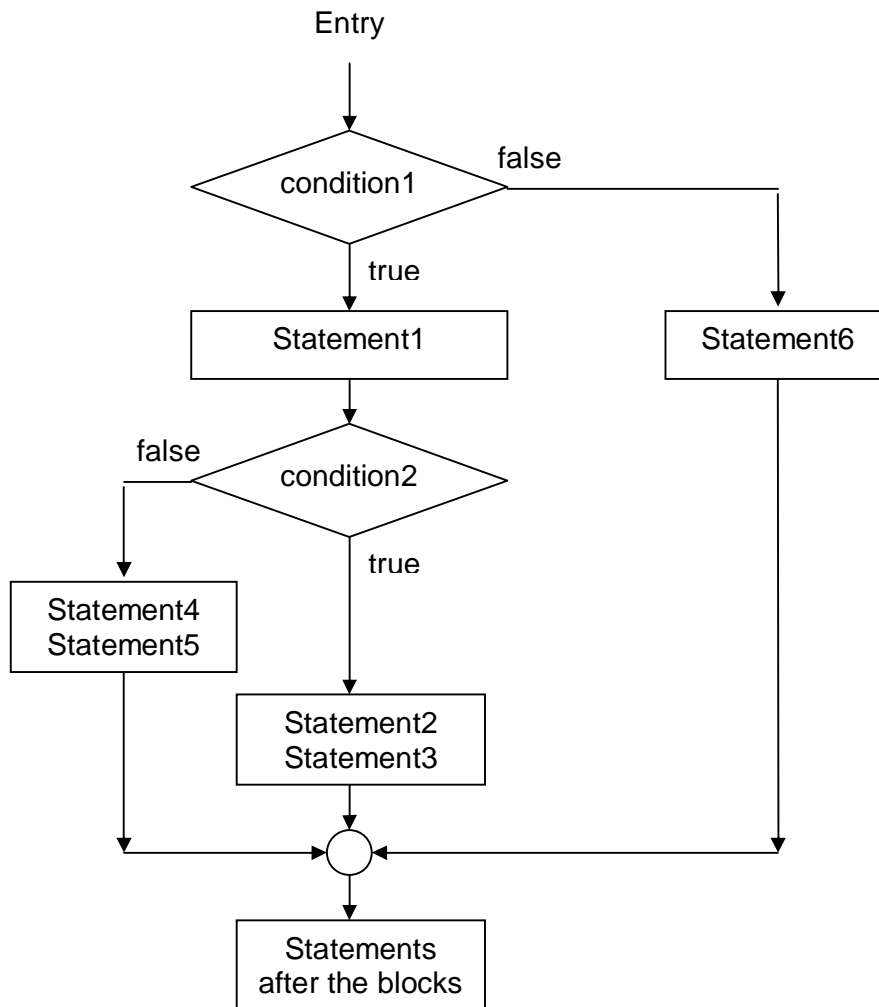
```

    }
}
else          //first 'else'
{
    statement6;
}
Statements after the if-else;

```

Here, if the condition1 is true statement1 is executed else statement6 is executed. Means, any one of the statement1 and statement6 is executed. Statement2 and statement3 are executed only when both condition1 and condition2 are true. Statement4 and statement5 are executed when condition1 is true and condition2 is false. We can say that in any case, maximum three statements are executed at a time and at least one. This hierarchy can be extended in any deep structure.

Flowchart of above structure can be shown in figure below.



**Flowchart of nested if-else structure**

Nesting of the if-else is useful in many conditions. Program 1.18 gives one of the practical applications of nested if-else.

```
//program to find largest of three
class Largest
{
    public static void main(String args[])
    {
        int x = 10, y = 15, z = 12;
        if(x > y)
        {
            if(x > z)
            {
                System.out.print(x);
            }
            else
            {
                System.out.print(z);
            }
        }
        else
        {
            if(y > z)
            {
                System.out.print(y);
            }
            else
            {
                System.out.print(z);
            }
        }
        System.out.println(" is largest");
    }
}
```

### Find largest number among three

Output:

15 is largest

This program finds the largest number among three integers. The conditions and statements are nested appropriately to find the output. In any case only one the statements is executed.

### The if-else ladder

The if-else ladder is another way of putting the 'if's together when multi-path decisions are involved. A multi-path decision is a chain of 'if's in which statements are associated with each 'else' is an 'if'. This construct is known as if-else ladder.

```

if(condition1)
    statement1;
else
    if(condition2)
        statement2;
    else
        if(condition3)
            statement3;
        else
            statement4;
            .
            .
            .

```

The conditions are evaluated from top of the ladder to the bottom. The very familiar example of this construct is to find the grade of the student according to marks. Such as,

For example examination board is giving grade according to marks as below:

Marks	Grade
Above 75	Distinction
60 to 74	First class
50 to 59	Second class
40 to 49	Pass class
Below 40	Fail

This can be done by applying the if-else ladder as,

```

if(marks>=75)
    grade = "distinction";
else
    if(marks >= 60)
        grade = "First class";
    else
        if(marks >= 50)
            grade = "Second class";
        else
            if(marks >= 40)
                grade = "Pass class";
            else
                grade = "fail";

```

This concept is demonstrated in program below.

```

//program to find grade according to marks
class IfElseLadder
{
    public static void main(String args[])
    {
        float marks = 62.12f;
    }
}

```

```

System.out.println("Marks : "+marks);
System.out.print("Your grade is : ");
if(marks >= 75)
    System.out.print("Distinction");
else
    if(marks >= 60)
        System.out.print("First class");
    else
        if(marks >= 50)
            System.out.print("Second class");
        else
            if(marks >= 40)
                System.out.print("Pass class");
            else
                System.out.print("Fail");
    }
}

```

### Simple if-else ladder

Output:

```

Marks : 62.12
Your grade is : First class

```

### The switch-case statement

[\[Asked in W'04'06\]](#)

Like C/C++ programming languages, switch is the multi-way selection statement. It tests the value of the variable (or expression) against the list of case values and when match is found, a block of statements associated with that case are executed. It is applicable when it is impossible to write the 'if' statement many times in the programs. Generally, it is used to create the menu driven program to select from the multiple choices.

General form of writing the switch-case statement is shown below:

```

switch(variable)
{
    case value-1:
        statements-1;
        break;
    case value-2:
        statements-2;
        break;
    - - - - -
    - - - - -
    default:
        default block;
}
statement-out;

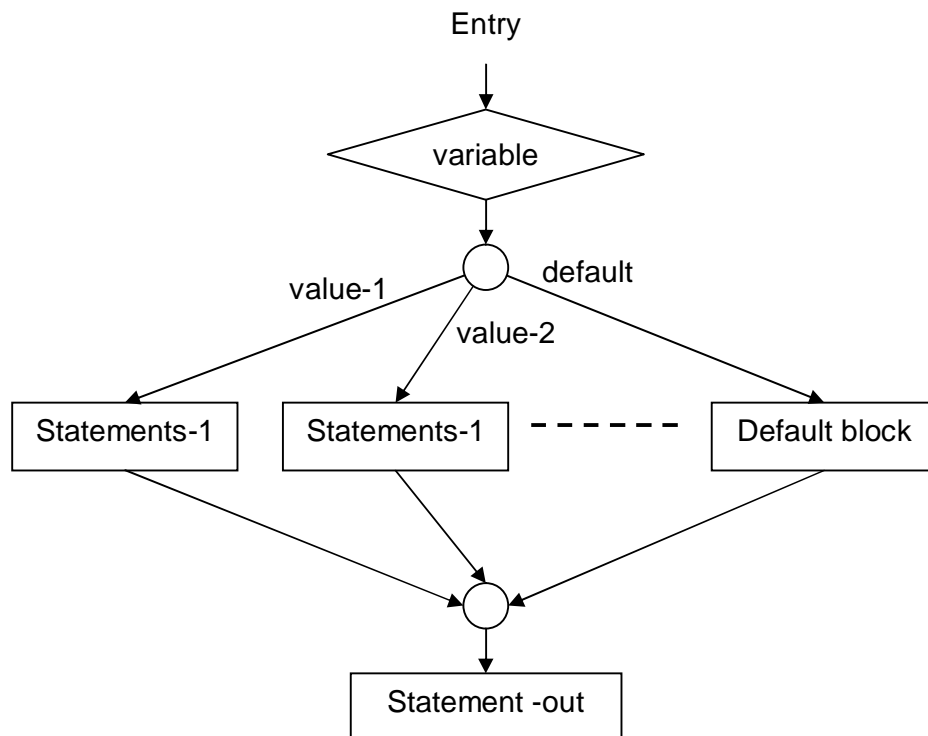
```



The variable written in switch brackets is either an integer or character variable or an expression which is evaluated to an integer value. value-1, value-2... are constants, which are also known as case labels. Each of this value is unique within switch statement. A colon (:) is given after each case label.

When the switch is executed, the value of the variable is successively compared with the values value-1, value-2.... If the case is found whose value matches with the variable then the statements in front of respective case are executed. The 'break' statement is written at the end of each case statement to signal the end of particular case. It causes the exit from the switch statement. It also transfers the program control out of the switch block to execute the statement-out.

The 'default' is executed when there is no match found in the cases. It is optional and can be written at any position in the switch block. But for simplicity, we are writing it at the end of switch.



**Flow-chart for switch-case statement**

Following programs demonstrates the use of switch-case statement.

```
//Program to display the number in character
class NumDisplay
{
    public static void main(String args[])
    {
        int x = 6;
        System.out.println("x = "+x);
    }
}
```

```

        System.out.print("It is ");
        switch(x)
        {
            case 1:  System.out.println("One");
                     break;
            case 2:  System.out.println("Two");
                     break;
            case 3:  System.out.println("Three");
                     break;
            case 4:  System.out.println("Four");
                     break;
            case 5:  System.out.println("Five");
                     break;
            case 6:  System.out.println("Six");
                     break;
            case 7:  System.out.println("Seven");
                     break;
            case 8:  System.out.println("Eight");
                     break;
            case 9:  System.out.println("Nine");
                     break;
            case 0:  System.out.println("Zero");
                     break;
            default: System.out.println("No. not correct");
        }
    }
}

```

### The switch-case statement

Output:

```

x = 6
It is Six

```

The program displays name of the number of the value of variable 'x'. Value of the variable 'x' is previously set to 6. When its value is matched with any of the constants written in front of case label the respective statements are executed. Always we will get the output as per the value of variable 'x'. At least the 'default' statement will be executed.

Observe another application based program.

```

// Menu driven program using switch-case.
import java.util.Scanner;
class Menu
{
    public static void main(String args[])
    {
        int choice, num;
        Scanner n = new Scanner(System.in);
        System.out.println("Menu....");
        System.out.println("1. Find positive");
    }
}

```

```

        System.out.println("2. Odd/Even");
        System.out.println("Enter the choice : ");
        choice = n.nextInt();
        switch(choice)
        {
            case 1:
                System.out.println("Enter number : ");
                num = n.nextInt();
                if(num > 0)
                    System.out.println("Positive...");
                else
                    System.out.println("Negative...");
                break;
            case 2:
                System.out.println("Enter number : ");
                num = n.nextInt();
                if(num%2 == 0)
                    System.out.println("Even...");
                else
                    System.out.println("Odd...");
                break;
            default:
                System.out.println("Wrong choice..");
        }
    }
}

```

### Application based switch-case program

Output1:

```

Menu....
1. Find positive
2. Odd/Even
Enter the choice :
1
Enter number :
-20
Negative...

```

Output2:

```

Menu....
1. Find positive
2. Odd/Even
Enter the choice :
2
Enter number :
63
Odd...

```

Remember: When the 'break' is not given after any case statement, all the cases are executed after that until a 'break' is not found. Try it.

## Decision making and looping

A Loop is the cycle of execution. It is what we can call repeatedly executing the same block of code until the termination condition is met. Java supports the looping feature which enables us to develop concise programs containing repetitive process without using unconditional branching statements.

In the process of looping, a sequence of statement is executed until some condition of termination of the loop is satisfied. Java is having three different loop statements.

1. while loop
2. do-while loop
3. for loop

### The while loop

This is simplest of all the looping structure. The structure of the while loop is given below.

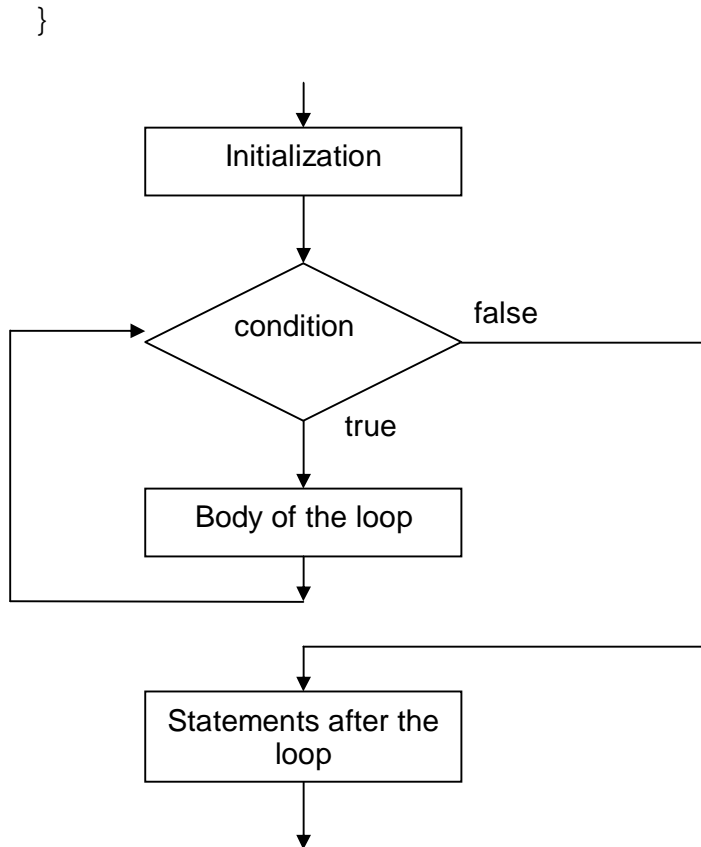
```
initialization;
while (condition)
{
    //body of the loop;
}
```

This can also be represented in the flowchart. View fig. 1.24.

The 'while' is an entry controlled loop structure. That is, before the program control enters the loop, the loop condition is checked. It can also be called as pre-test loop. If the condition given is evaluated to true then the body of the loop is executed until the condition satisfies to true. When the condition becomes false, the loop is terminated. Then statements written after the loop are executed. If condition doesn't become true any time then loop will go into infinite iteration execution.

Body of the loop may contain any number of statements. If the loop contains only one statement then the curly braces are not required. Condition given may contain combination of relational and logical operators as well as boolean variable variables too. For example, if we want to print "I Love Java" for 10 times. It can be written in following form.

```
int i = 0;
while(i<10)
{
    System.out.println("I Love Java");
    i++;
}
```



**Flowchart of while loop statement**

Here, the value of variable *i* is initialized to 0 in declaration statement. In the while loop we are changing / incrementing the value of it and checking the condition whether it reaches to 10 or not. In short, the loop can be executed for 10 times. So, variable '*i*' is called as loop counter because it controls the execution of the loop. The same output can also be obtained by following statements.

1. 

```
int i = 10;
while(i>0)
{
    System.out.println("I Love Java");
    i--;
}
```
2. 

```
int i = 1;
boolean b = true;
while(b)
{
    System.out.println("I Love Java");
    i++;
    if(i>=10)
        b = false;
}
```

```
}
```

Any of the above can be applied to find the required output. So the program design is depending upon the programmer and his logic. Your algorithm can also be different.

Observe the following program which finds the addition of all the natural numbers from 1 to 10.

```
//Program to find addition of 10 natural numbers.
class WhileAdd
{
    public static void main(String args[])
    {
        int a = 0, sum = 0;
        while(a<=10)
        {
            sum = sum + a;
            a++;          // or a = a+1;
        }
        System.out.println("Addition is "+sum);
    }
}
```

### **Program to find addition of 10 natural numbers**

Output:

Addition is 55

### **The do-while loop**

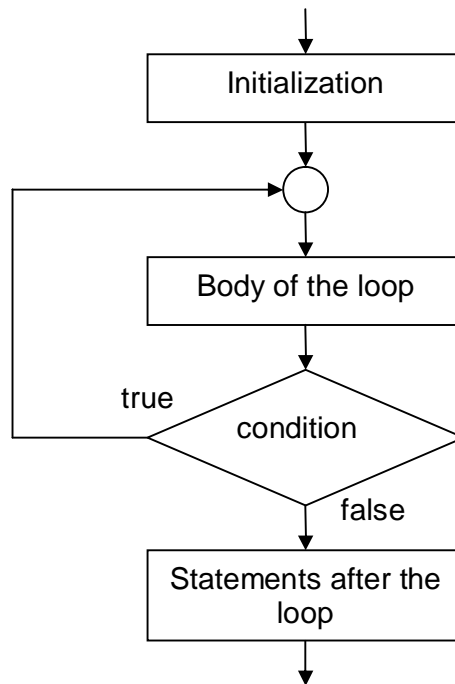
Sometimes in the program it is desirable to execute your loop at least once. The do-while loop can be applied in this case. It is post-test loop. That is, the loop is executed first and then the loop condition is checked. The condition is written at the end of the loop. This is also called as exit-controlled loop. That is, exit of the program control from loop is decided by loop condition.

The syntax of do-while loop statement is,

```
do
{
    //body of the loop;
}
while(condition);
```

On reaching the do statement, the body of the loop is executed first. Then the loop condition written in 'while' is checked. If it is true then program control will proceed to execute next iteration of the loop else next statements are executed. When the condition becomes false, the

loop gets terminated. Remember, there is a terminating semicolon given at the end of the while statement. Do not miss that.



**Flowchart of while loop statement**

Consider the example,

```

int a = 0;
do
{
    System.out.println("I Love Java");
    a++;
}
while(a<10);
  
```

This will give the same output as above first example of the while loop gives to print "I Love Java" for 10 times. Here, if we change the value of variable 'a' to any other value then also the loop will execute at least once.

Observe another example of do-while which prints the squares of numbers from 1 to 10.

```

//Program squares of numbers
class DoWhileSquare
{
    public static void main(String args[])
    {
        int num = 1;
        do
        {
  
```

```
        System.out.print("Square of ");
        System.out.print(num+" : ");
        System.out.println(num*num);
        num++;
    }
    while(num<=10);
}
```

### Program using simple do-while loop

Output:

```
Square of 1 : 1
Square of 2 : 4
Square of 3 : 9
Square of 4 : 16
Square of 5 : 25
Square of 6 : 36
Square of 7 : 49
Square of 8 : 64
Square of 9 : 81
Square of 10 : 100
```

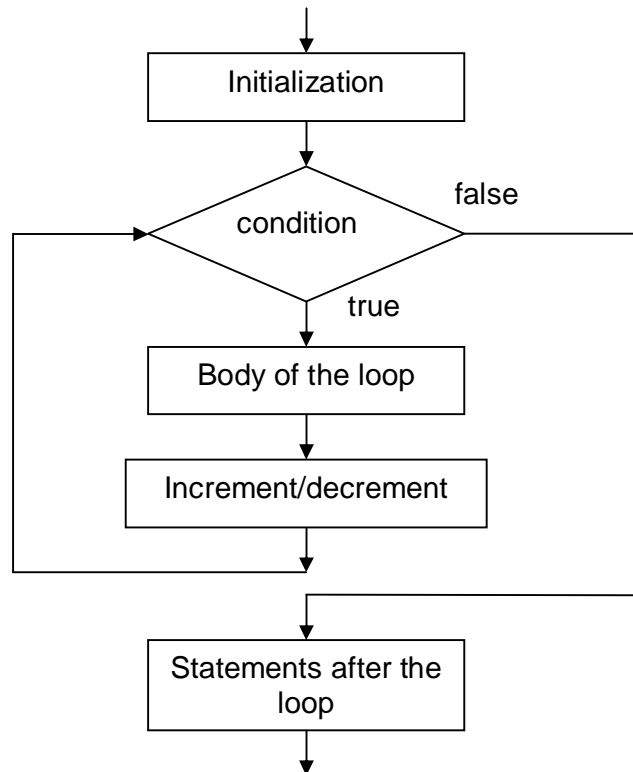
### The for loop

The 'for' loop is another entry-controlled loop, that provides a more concise loop control structure. That is, initialization, condition and increment/decrement can be done in the single loop statement as given below.

```
for(initialization; condition; increment/decrement)
{
    Body of the loop;
}
```

Operation of the 'for' loop is shown in the flowchart





**Fig. 1.26 Flowchart of the 'for' loop**

This loop control structure works as below.

1. Initialization part is executed first. Here, we can do the initialization of the loop control variables. This statement is executed only once i.e. before the start of execution of the loop.
2. In the second stage, the condition given is tested. If it is evaluated to true, then body of the loop is executed. Generally, the condition is related with the loop counter. But it is not necessary to give condition related to loop counter only. The condition is tested for each iteration of the loop. When it becomes false the loop gets terminated.
3. After completion of execution of the body of the loop, increment / decrement statement is executed. Any action we can write in this statement. Then again control gets transferred to the condition to execute the next iteration.

For example, consider the following segment.

```

for(a = 0; a < 10; a++ )
{
    System.out.println(a);
}
  
```

Here the loop counter 'a' is initialized to 0 when loop execution started. The loop is executed for 10 numbers of times by incrementing

the value of counter by 1. Each time the value of variable 'a' gets printed onto the screen. When the body of the loop contains only one statement then it is not necessary to give the curly braces. Same statement can also be written as,

```
for(a = 0; a < 10; a++ )
    System.out.println(a);
```

Loop can contain the counters, whose value is decrementing such as,

```
for(a = 10; a > 0; a-- )
    System.out.println(a);
```

In this case, the loop counter variable 'a' is initialized to 10 and then it is get decremented by one, on iteration until its value becomes less than 0. Following table shows the comparison among all three loops discussed above.

while	do-while	for
<pre>x = 0; while(x&lt;10) {     -----;     -----;     x++; }</pre>	<pre>x = 0; do {     -----;     -----;     x++; } while(x&lt;10);</pre>	<pre>for(x=0;x&lt;10;x++) {     -----;     -----; }</pre>

**Table 1.16 Comparison of three loop structures**

Observe the following example of the 'for' loop which finds factorial of the number.

```
//Find factorial of number using for loop
import java.util.Scanner;
class ForFact
{
    public static void main(String args[])
    {
        int fact = 1, num;
        Scanner in = new Scanner(System.in);
        System.out.print("Enter number : ");
        num = in.nextInt();
        for(int i = num ;i>0; i--)
            fact = fact * i;
        System.out.println("Factorial : "+ fact);
    }
}
```

## Finding factorial using 'for' loop

Output:

```
Enter number : 5
Factorial : 120
```

We know that factorial of the number 'n' can be obtained using formula,

$$n * (n-1) * (n-2) \dots 1$$

So, we have started our loop counter from 'n' itself and decremented it by one until its value becomes less than 1. For storing the values we have taken one variable 'fact' which is initialized to 1 at the start. After all, in the loop we have just multiplied the loop counter in the variable 'fact'. At the end of the loop, the 'fact' will contain final value of factorial of the number 'n'.

### Additional features of the 'for' loop

The 'for' loop have several additional capabilities which has made is more beneficial over other looping controls.

1. More than one initialization statements can be written in the 'for' statement at a time. But they must be separated by a comma.

For example,

```
int x = 10;
for(y=0;y<10;y++)
```

this can also be written as,

```
for(x=10, y=0;y<10;y++)
```

2. A variable can be declared and initialized in the 'for' loop itself. But its scope remains within that loop only.

For example,

```
for(int y=0;y<10;y++)
```

here the variable 'y' can only be used in the respective 'for' loop only. There is no scope and visibility of this variable outside of the loop.

3. The condition part can also have combination of the conditions using logical operators which returns values true or false. They can not be separated using comma as in the initialization section.

For example,

```
for(int i=0, a = 10;i<10 && a >0;a++)
```

in this example if both of the conditions are evaluated to true then only the loop is executed.

4. Increment / decrement section can have multiple statements separated by a comma. Such as,

```
int x = 5;
for(int a = 10;a<100 || x > 0; a++, y--)
```

here, on each iteration of the loop, value of 'a' is incremented and 'y' is decremented as shown in the last section of the 'for' loop.

5. Any of the statements of initialization, condition and increment / decrement can be eliminated by keeping it blank. For example,

```
a = 12;
for(;a<100;)
{
    a++;
    System.out.println("Welcome");
}
```

6. All the statements in the loop can be eliminated just by keeping two semicolons in the brackets such as,

```
for(; ;)
{
}
```

but this loop will run infinitely.

7. When the following statement is executed.

```
for(int x = 0;x<10;x++);
```

compiler will not show any error. This is called as empty statement. Many times by mistake we give semicolon at the end. One must take care of that. But in several cases this feature is most applicable. We will see it in upcoming chapters.

8. See one loop statement in programming languages such as C/C++.

```
for(int a = 10;a;a--)
```

this statement of the loop will work properly in these languages. It will be executed for 10 times. That is, it will be executed until value of the variable 'a' reaches to zero. In Java, this concept is not supported. Because C/C++ supposes the returned value of the condition is an integer. For them a value is same as 'true' and zero means 'false'. But in Java, the condition must be evaluated to boolean variable 'true' and 'false'.

## Nesting of the loop controls

Writing one loop in another is supported by Java. That is we can nest the loops. For example,

```

-----;
-----;
for(int x = 0;x<10;x++)
{
    y = 0;
    while(y<10)
    {
        -----;
        -----;
    }
}
-----;
-----;

```

Here, the 'for' loop is called as outer loop and 'while' loop is referred as inner loop. For every iteration of the outer loop, all the iterations of inner loops are completed. Nesting of the loops is depending upon the application.

For example, we want to print the following output using 'for' loop.

```

* * * * *
* * * *
* * *
* *
*

```

In this case, the nesting of the loop is must. That is, the outer loop will have a decrement counter whose value starts from five and moves to zero. And the inner loop will use this value to print that specific number of asterisks. This has been solved in the following program.

```

//Program to print given asterisks \[Asked in W'04\]
class NestingStar
{
    public static void main(String args[])
    {

```

```

        int x, y;
        for(x=5;x>0;x--)
        {
            for(y=x;y>0;y--)
                System.out.print("* ");
            System.out.println();
        }
    }
}

```

**Program 1.29 Program to print given asterisks**

Output:

```

* * * * *
* * * *
* * *
* *
*

```

## Jump statements

Sometimes while executing the loop it is necessary to skip some part of the loop or exit from the loop when certain conditions are met. In these cases the jump statements are used. There are two jump statements related to loop structures.

1. break statement.
2. continue statement.

### The break statement

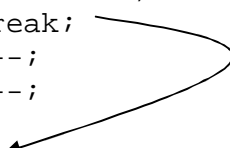
We have already studied the break statement in switch-case structure. The 'break' can also be used in the loop control structures. It is used to jump out of the loop when this statement is encountered inside the loop. Generally, the 'break' statement is associated with an 'if' statement. That is, when the condition is satisfied, the jump will occur.

For example,

1.
 

```

while(condition)
{
    -----;
    -----;
    if(condition)
        break;
    -----;
    -----;
}
-----;
            
```


2.
 

```

do
{

```

```

    -----;
    -----;
    if(condition)
        break;
    -----;
    -----;
} while(condition);
-----;

```

3. for(;;)

```

{
    -----;
    -----;
    if(condition)
        break;
    -----;
    -----;
}
-----;

```

4. for(;;)

```

{
    -----;
    while(condition)
    {
        if(condition)
            break;
        -----;
        -----;
    }
    -----;
}
-----;

```

Note: When the 'break' statement is used in the nested loops. It will only exit the loop in which it is actually encountered.

Following program demonstrates the use of break statement.

```

//demonstration of break statement
class BreakDemo
{
    public static void main(String args[])
    {
        int var = 10;
        while(var<20)
        {
            System.out.println(var);
            var = var + 1;
            if(var>15)

```

```

        break;
    }
    System.out.println("Loop completed...");
}
}

```

### Demonstration of break statement

Output:

```

10
11
12
13
14
15
Loop completed...

```

### Using break as a form of goto / labeled loops

[\[Asked in S'04\]](#)

In addition to its uses with the switch statement and loops, the break statement can also be used to provide a “civilized” form of the goto statement. Java does not have a goto statement, because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations. There are, however, a few places where the goto is a valuable and legitimate construct for flow control. For example, the goto can be useful when we 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, we can break out of one or more blocks of code. These blocks need not be part of a loop or a switch. They can be any block. Further, you can specify precisely where execution will resume, because this form of break works with a label. As you will see, break gives you the benefits of a goto without its problems.

The general form of the labeled break statement is,

```
break label;
```

Here, label is the name of a label that identifies a block of code. When this form of break executes, control is transferred out of the named block of code. The labeled block of code must enclose the break statement, but it does not need to be the immediately enclosing block. This means that you can use a labeled break statement to exit from a set of nested blocks. But you cannot use break to transfer control to a block of code that does not enclose the break statement.

Observe the following example,

```

// Using break as a form of goto.
class BreakGoto
{

```



```

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

```

### Using break as a form of goto.

Output:

```

Before the break.
This is after second block.

```

In this example, three different blocks are created with three different names i.e. first, second and third. When the 'break second;' statement is executed, the control gets transferred out of the block named 'second'. Any further statement will not be executed. The following program will explain how the break with label is used to break the outer loop, which is not possible using general break statement.

```

//program to demonstrate the break
class BreakNested
{
    public static void main(String args[])
    {
        outer:
        for(int x=0;x<10;x++)
        {
            System.out.print("This is "+x+": ");
            for(int y=0;y<10;y++)
            {
                if(y==5)
                    break outer;
                System.out.print(" "+y+" ");
            }
        }
        System.out.println("Program finished...");
    }
}

```

```

    }
}

```

### Program to demonstrate the break

Output:

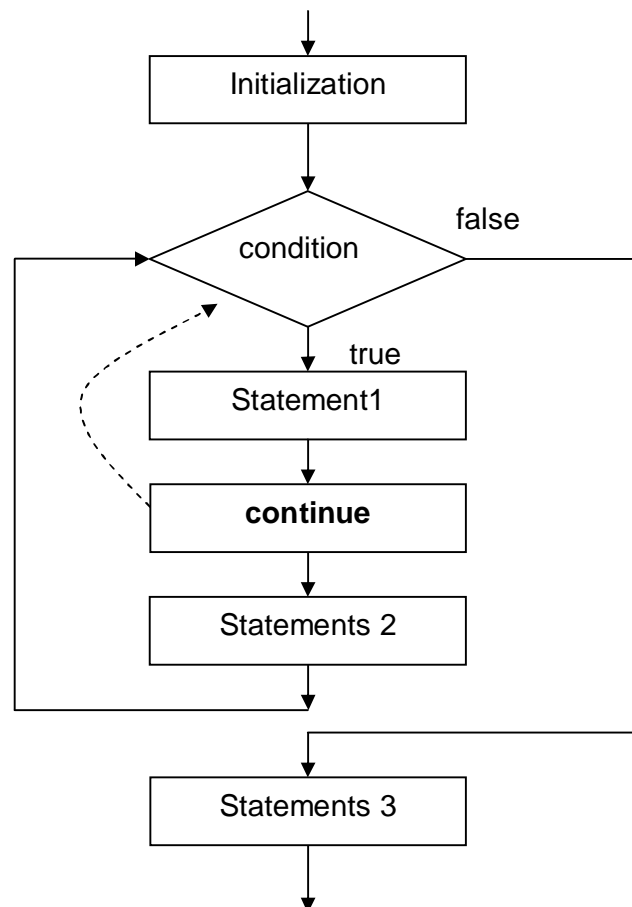
```
This is 0:  0  1  2  3  4 Program finished...
```

### The continue statement

It is useful to force the early iteration of the loop. When we want to continue the next iteration of the loop by skipping some part inside it, the 'continue' statement can be used. It is also associated with the condition. General form of the 'continue' is:

```
continue;
```

When the 'continue' is encountered in the loop the program control skips all the further statements after that in the loop and proceed to execute the next iteration. Many programmers get confused with 'continue' so better way the following flowchart will explain the use of 'continue' statement.



### Flowchart of 'continue' statement

The dotted lines in the flowchart shows when continue gets executed where the program control get transferred. Following program will explain a small practical application of the 'continue' statement.

```
// Demonstration of the 'continue' statement.
class ContinueDemo
{
    public static void main(String args[])
    {
        for(int i=0; i<10; i++)
        {
            if (i%2 == 0)
                continue;
            System.out.println(i + " ");
        }
    }
}
```

### Demonstration of the 'continue' statement

Output:

```
1
3
5
7
9
```

In this program, the 'continue' is executed upon the occurrence of the loop counter as the even value. So the output statement will not execute on this occasion. Thus, the program outputs only the odd numbers on the screen.

As with the break statement, continue may specify a label to describe which enclosing loop to continue. See the example program that uses continue to print a triangular multiplication table from 0 through 9.

```
// Using continue with a label.
class ContinueLabel
{
    public static void main(String args[])
    {
        outer: for (int i=0; i<10; i++)
        {
            for(int j=0; j<10; j++)
            {
                if(j > i)
                {
                    System.out.println();
                    continue outer;
                }
            }
        }
    }
}
```

```
        System.out.print(" " + (i * j));  
    }  
    }  
    System.out.println();  
}  
}
```

### Using continue with a label

Output:

```
0  
0 1  
0 2 4  
0 3 6 9  
0 4 8 12 16  
0 5 10 15 20 25  
0 6 12 18 24 30 36  
0 7 14 21 28 35 42 49  
0 8 16 24 32 40 48 56 64  
0 9 18 27 36 45 54 63 72 81
```

-----