## Introduction to Object Oriented Programming

→→ **Introduction:** One Characteristic that is constant in the software industry today is the "change". Since the invention of the computer, many programming approaches have been tried. These include techniques such as modular programming, top-down programming, bottom-up programming and structured programming. The primary motivation in each case has been the concern to handle the increasing complexity of programs that are reliable and maintainable.

With the advent of languages such as C, structured programming became very popular and was the paradigm of the 1980's. Structured programming became very powerful tool that enabled programmers to write moderately complex programs fairly easy. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Object-Oriented Programming (OOP) is an approach to program organization and development, which attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several new concepts. Java is a pure object – oriented language.

→→ **Need of OOP:** Two Paradigms of Programming: All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around what is happening and others are written around who is being affected. These are the two paradigms that govern how a program is constructed.

- The first way is called the process-oriented model. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success. Problems with this approach appear as programs grow larger and more complex. To manage increasing complexity, the second approach, called object-oriented programming, was conceived.

- Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined methods to that data. An object-oriented program can be characterized as data controlling access to code.

**Data Abstraction:** Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction.

→→ **Principles of OOP:** The following are the 3 principles of OOP

1. Encapsulation
2. Inheritance
3. Polymorphism

1. Encapsulation: Wrapping up of the data and methods into a single unit is known as Encapsulation. Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

   In Java the basis of encapsulation is the class. A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality.

2. Inheritance: Inheritance is the process by which objects of one class acquire the properties of objects of another class. A class inherits state and behavior from its superclass. Inheritance provides a powerful and natural mechanism for organizing and structuring software programs.

   Reusability is main advantage in inheritance by which we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.

3. Polymorphism: Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.

→ **Dynamic Binding**: Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic Binding is that the code associated with a given procedural call is not known until the time of call at runtime. Dynamic Binding is associated with polymorphism and inheritance.

→ **Message Communication:** An OOP consists of a set of objects that communicate with each other. The processing of programming in an OO language involves the following steps:

1) Creating classes that define objects and their behavior.
2) Creating Objects form class definitions.
3) Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people messages to one another.

**→→ Procedure oriented Programming Vs Object Oriented Programming**

**→ Procedure oriented Programming:**

- In this approach, the problem is always considered as a sequence of tasks to be done. A number of functions are written to accomplish these tasks. Here primary focus on Functions and little attention on data.
- There are many high level languages like COBOL, FORTRAN, PASCAL, C used for conventional programming commonly known as Procedure oriented Programming.
- Procedure oriented Programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions.

**Characteristics of POP:**

- Emphasis is on doing actions.
- Large programs are divided into smaller programs known as functions.
- Most of the functions shared global data.
- Data move openly around the program from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

→ **OOP:** OOP allows us to decompose a problem into a number of entities called objects and then builds data and methods around these entities.

**Def:** OOP is an approach that provides a way of modularizing programs by creating portioned memory area for both data and methods that can used as templates for creating copies of such modules on demand.

**OOP Characteristics:**

- Emphasis on data.
- Programs are divided into what are known as methods.
- Data structures are designed such that they characterize the objects.
- Methods that operate on the data of an object are tied together.
- Data is hidden.
- Objects can communicate with each other through methods.
- Reusability.
- Follows bottom-up approach in program design.

→→**Applications of OOPs:**

- Real time systems
- Simulation and modeling
- OO database
- Hypertext, hypermedia and expert-text
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CAM/CAD systems.

**→Advantages of OOPs:**

- This eliminates redundant code and extends the use of classes with the concept of inheritance.
- This can build the programs from the standard working modules that communicate with one another, rather than having to start writing the code from beginning. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instance of an object to exist without any interference.
- Software complexity can be managed.
- OO systems can be easily upgraded from small to large systems.

**→→ History of JAVA:** Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades.

By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed, for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs. However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. Within a few years, the World Wide Web and the Internet would reach critical mass. This event would precipitate another revolution in programming.

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

- It should use the object-oriented programming methodology.
- It should allow the same program to be executed on multiple operating systems.
- It should contain built-in support for using computer networks.
- It should be designed to execute code from remote sources securely.
- It should be easy to use by selecting what was considered the good parts of other object oriented languages.

The Creation of Java: Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995.

Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Ho, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Java was designed not for the Internet, the primary motivation was the need for a platform-independent (that is, architecture- neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost efficient—solution was needed.

In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

Because of the similarities between Java and C++, Java should not be considered as the "Internet version of C++." Java has significant differences. Even though 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. 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.

→ **Java and Internet:** The Internet helped Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the ***applet*** that changed the way the online world thought about content. Java also addressed some of the issues associated with the Internet: portability and security.

1. **Java Applets:** An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace.
2. **Security:** As you are likely aware, every time you download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. Java achieved this protection by conning an applet to the Java execution environment and not allowing it access to other parts of the computer.
3. **Portability:** Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to

enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The same code must work on all computers.

**→→ Java Virtual Machine:**

**Byte Code:** The key that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code. Rather, it is byte code.

Byte code is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). In essence, the original JVM was designed as an interpreter for byte code.

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

A Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language.

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

**Just-In-Time (JIT) compiler:** When a JIT compiler is part of the JVM, selected portions of byte code are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. It is used to improve the performance.

→→ **Java Features:** The Java language was invented not only for portability and security, but also there are many other features played an important role. The following are the features of java

a) **Simple, Small and Familiar:** Java is a simple and familiar language because it contains many features of other Languages like c and C++. Java removes complexity because it doesn't use pointers, storage classes and go to statements and also java doesn't support Multiple Inheritance and operator overloading.

b) **Compiled and Interpreted:** Java combines the features of both compiled and interpreted approaches. First, Java compiler translates source code into what is known as byte code instructions. Byte codes are not machine instructions. Second, Java interpreter generates machine code that can be directly executed by the machine that is running the java program. Java is both compiled and interpreted language.

c) **Platform-Independent and Portable:** Java programs can be easily moved from one computer system to another, anywhere and anytime. Changes and upgrades in operating systems, processors and system resources will not force any changes in java programs. Java ensures portability in two ways. First, Java compiler generates byte code instruction that can be implemented on any machine. Secondly, the sizes of the primitive data types are machine independent.

d) **Object-Oriented:** Java is a true object-oriented language. Almost everything in java is an object. All program code and data reside within objects and classes. Java comes with an extensive set of classes, arranged in packages.

e) **Robust and Secure:** Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. It is designed as a garbage-collected language relieving the programmers virtually all memory

management problems. Java also incorporates the concepts of exception handling which captures series errors and eliminates any risk of crashing the system. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in java ensures that programs cannot gain access to memory locations without proper authorization.

f) **Distributed:** Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on Internet as easily as they can do in local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

g) **Multithreaded and Interactive:** Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programming. This means that we not wait for the application to finish one task before beginning another. The java runtime comes with tools that support multiprocess synchronization and construct smoothly running interactive systems.

h) **High Performance:** Java performance is impressive for an interpreted language, mainly due to the use of intermediate byte code. Java speed is comparable to the native C/C++. Java architecture is also designed to reduce the overheads during runtime. Further, the incorporation of multithreading enhances the overall execution speed of java programs.

i) **Dynamic and Extensible:** Java is a dynamic language. Java is capable of dynamically linking a new class libraries, methods, and objects. Java can also determine the type of class through a query, making it possible to dynamically link or abort the program, depending on the response. Java programs support functions written in other languages such as C and C++. These functions are known as native methods. Native methods are linked dynamically at runtime.

→→ **Java Program Structure:** A java program may contain many classes of which only one class defines a main method. Classes contain data members and methods that operate on the data members of the class. Methods may contain data type declarations and executable statements. The Java program contains the sections as in following figure.

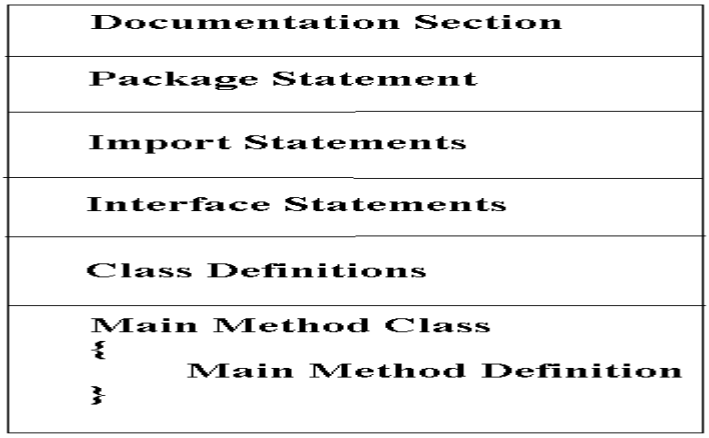


Fig: General Structure of a Java Program

a) **Documentation Section:** This Section comprises a set of comment lines giving the name of the program, the author and other details. Comments must explain why and what of classes. Java uses three styles of comments. One is ***single-line comment (//),*** Second one is ***multi-line comment (/* … */)*** and the third one is ***documentation comment (/** … */).***
b) **Package Statement:** The first statement allowed in a java file is a package statement. This statement declares a package name and informs the compiler that the classes defined here belong to this package. Example: ***package student;*** The Package statement is optional.
c) **Import Statements:** After a package statement there can be any number of import statements. This is similar to the ***#include*** statement in C. Example: ***import java.util.Scanner;***. This statement instructs the interpreter to load the ***Scanner*** class contained in the package ***java.util***. Using import statements, we can have access to classes that are part of other named packages.
d) **Interface Statements:** An interface is like a class but includes a group of method declarations. This is an optional section.

e) **Class Definitions:** A java program may contain multiple class definitions. Classes are the primary and essential elements of a java program. These classes are used to map the objects of real-world problems.

f) **Main Method Class:** Every stand-alone program requires a ***main*** method as its starting point; this class is the essential part of a java program. The main method creates objects of various classes and establishes communications between them. On reaching the end of ***main***, the program terminates and the control passes back to the operating system.

## Programming Constructs

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

**Declaring a Variable,** in java all variables must be declared before they can be used.

***Syntax:*** *type identifier* [= *value*] [, *identifier* [= *value*] ...];

The *type* is one of java's atomic types, or the name of a class or interface. The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. That the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use comma separated list.

**E.g.:** int a, b, c; or int a = 3, b = 5, c; byte d = 22; double pi = 3.14159; char s='s';

**Dynamic Initialization:** Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

```
//Dynamic initialization.
import java.util.Scanner;
class DynInit
{
        public static void main(String args[])
        {
                int num1,num2,num3;
```

```
            Scanner s=new Scanner(System.in);
            System.out.println("Enter num1 & num2 values");
            num1=s.nextInt();
            num2=s.nextInt();
            num3=num1+num2;
            System.out.println("num3 value is " + num3);
        }
}
```

→→ **Data Types:** Java defines 8 types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. Data types can be divided into 2 types

1. Primitive Data Types (**byte**, **short**, **int**, **long**, **float**, **double, char, boolean**)
2. Non- Primitive Data Types (**class, String, arrays, interfaces**)

→ **Primitive Data types:** The primitive types are defined to have an ***explicit range*** and mathematical behavior.

1. **Integers:** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers, positive & negative numbers.
2. **Floating-point numbers:** This group includes **float** and **double**, which represent numbers with fractional precision.
3. **Characters:** This group includes **char**, which represents symbols in a character set, like letters and numbers.
4. **Boolean:** This group includes **boolean**, which is a special type for representing true/false values.

1. **Integers:** Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values.
   a. Java does not support unsigned, positive-only integers.
   b. Many other computer languages support both signed and unsigned integers.
   c. Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.
   d. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them.

**Width and ranges of integer types**

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

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

ii. ***short:*** **short** is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used in Java. **E.g.: short** s;

iii. ***int:*** int is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Therefore, **int** is often the best choice when an integer is needed. **E.g.: int** a, b, c;

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

**2. Floating point numbers:** Floating point numbers are also known as real numbers. These are used for representing the fractional numbers. There are two types **float** and **double**

**Width and ranges of floating types**

| Name | Width(bits) | Approximate Range |
|---|---|---|
| float | 32 | 1.4e-045 to 3.4e+038 |
| double | 64 | 4.9e-324 to 1.8e+308 |

i. **float:** float specifies a single precision value that uses 32 bits of storage. Float is useful when you need a fractional component. **E.g.: float** a, b;

ii. **double:** Its representation is faster than the float representation. It can be used for high speed calculations. **E.g.: double** a, b, c;

3. **Characters:** In Java, the data type used to store characters is **char**. **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, Java uses Unicode to represent characters. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. **E.g.: char** ch1, ch2; ch1 = 'Y'; ch2 = 88; // code for X;
4. **Booleans:** Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**. **boolean** is also the type ***required*** by the conditional expressions that govern the control statements such as **if** and **for**.

**E.g.: Program**

```
class BoolTest
{
        public static void main(String args[])
        {
                boolean b;
                b = false;
                System.out.println("b is " + b);
                b = true;
                System.out.println("b is " + b);
                // a boolean value can control the if statement
                if(b)
                        System.out.println("This is executed.");
                b = false;
                if(b)
                        System.out.println("This is not executed.");
                // outcome of a relational operator is a boolean value
                System.out.println("10 > 9 is " + (10 > 9));
        }
}
```

**Output:**

```
b is false
```

b is true
This is executed.
10 > 9 is true.

→ **Non-Primitive (Reference) Data type:** A reference data type is used to refer to an object. A reference variable is declared to be of specific and that type can never be change.

→→ **Identifiers: -** Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore _.

a. Identifiers cannot start with a number.
b. After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
c. There is no limit to the number of characters an identifier can contain.
d. You can't use a Java keyword as an identifier.
e. Identifiers in Java are case-sensitive; foo and FOO are two different identifiers.
f. A legal identifier for a variable is also a legal identifier for a method or a class.

**Naming Conventions:** Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc. But, it is not forced to follow. So, it is known as convention not rule. All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention. **Advantage of naming conventions in java:** By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

| **Name** | **Convention** |
|---|---|
| class name | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| interface name | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| method name | should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |

| | |
|---|---|
| variable name | should start with lowercase letter e.g. firstName, orderNumber etc. |
| package name | should be in lowercase letter e.g. java, lang, sql, util etc. |
| constants name | should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc. |

→→ **Keywords:** The Java programming language has total of 50 reserved keywords which have special meaning for the compiler and cannot be used as variable names. Following is a list of Java keywords in alphabetical order, click on an individual keyword to see its description and usage example.

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

**Points regarding Java keywords:**

- **const** and **goto** are resevered words but not used.
- **true**, **false** and **null** are literals, not keywords.
- all keywords are in lower-case.

| **Category** | **Keywords** |
|---|---|
| *Access modifiers* | **private**, **protected**, **public** |
| *Class, method, variable modifiers* | **abstract**, **class, extends, final, implements, interface, native,new, static, strictfp, synchronized, transient, volatile** |
| *Flow control* | **break, case, continue, default, do, else, for, if, instanceof,return, switch, while** |
| *Package control* | **import, package** |

| | |
|---|---|
| *Primitive types* | **boolean, byte, char, double, float, int, long, short** |
| *Error handling* | **assert, catch, finally, throw, throws, try** |
| *Enumeration* | **enum** |
| *Others* | **super, this, void** |
| *Unused* | **const, goto** |

→→ **Literals:** Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables. Literals can be any number, text, or other information that represents a value. Java language specifies five major types of literals. They are: Integer literals, Floating literals, Character literals, String literals, and Boolean literals. Each of them has a type associated with it. The type describes how the values behave and how they are stored.

1. **Integer literals:** Integer data types consist of the following primitive data types: int, long, byte, and short. byte, int, long, and short can be expressed in decimal(base10), hexadecimal(base 16) or octal(base 8) number systems as well. Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals.
   Examples: int decimal = 100; int octal = 0144; int hexa = 0x64;

2. **Floating-point literals:** Floating-point numbers are like real numbers in mathematics, for example, 4.13179, -0.000001. Java has two kinds of floating-point numbers: float and double. The default type when you write a floating-point literal is double, but you can designate it explicitly by appending the D (or d) suffix. However, the suffix F (or f) is appended to designate the data type of a floating-point literal as float. We can also specify a floating-point literal in scientific notation using Exponent (short E ore), for instance: the double literal 0.0314E2 is interpreted as: $0.0314 * 10^2$ (i.e. 3.14).

   ```
   float ff = 89.0f; double dou = 89.0D; double doub = 89.0d;
   float f = 89.0; // Type mismatch: cannot convert from double to float
   double doubl = 89.0; //OK, by default floating point literal is double
   ```

3. **Boolean Literals:** The values true and false are treated as literals in Java programming. When we assign a value to a boolean variable, we can only use these two values. Unlike C,

we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java. We have to use the values true and false to represent a Boolean value. Example boolean chosen = true;

4. **Null Literals:** The final literal that we can use in Java programming is a null literal. We specify the Null literal in the source code as 'null'. To reduce the number of references to an object, use null literal. The type of the null literal is always null. We typically assign null literals to object reference variables. For instance s = null;
5. **Character literals:** char data type is a single 16-bit Unicode character. We can specify a character literal as a single printable character in a pair of single quote characters such as 'a', '#', and '3'. You must know about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuation etc. Below table shows a set of these special characters.

| Escape | Meaning |
|---|---|
| \n | New line |
| \t | Tab |
| \b | Backspace |
| \r | Carriage return |
| \f | Formfeed |
| \\ | Backslash |
| \' | Single quotation mark |
| \" | Double quotation mark |
| \d | Octal |
| \xd | Hexadecimal |
| \ud | Unicode character |

If we want to specify a single quote, a backslash, or a non-printable character as a character literal use an escape sequence. An escape sequence uses a special syntax to represents a character. The syntax begins with a single backslash character.

6. **String Literals:** The set of characters in represented as String literals in Java. Always use "double quotes" for String literals. There are few methods provided in Java to combine strings, modify strings and to know whether to strings have the same values.

| | |
|---|---|
| "" | The empty string |
| "\"" | A string containing |
| "This is a string" | A string containing 16 characters |
| "This is a " + "two-line string" | actually a string-valued constant expression, formed from two string literals |

**Standard Default Values:** In java, every variable has a default value. If we don't initialize a variable when it is first created, java provides default value to that variable type automatically as shown in following table:

| Type of Variable | Default Value |
|---|---|
| Byte | 0 (byte) |
| Short | 0 (short) |
| int | 0 |
| Long | 0L |
| Float | 0.0f |
| Double | 0.0d |
| Char | null character |
| Boolean | false |
| Reference | null |

→→ **Operators:** Java provides a rich set of operators. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables.

The following table shows the related categories:

| Category | Operators |
|---|---|
| Simple assignment | = |
| Arithmetic | + - * / % |
| Unary | + - ++ -- ! |
| Relational | == != > >= < <= |
| Conditional | && \|\| ? : (ternary) |
| Type comparison | instanceof |

| | |
|---|---|
| Bitwise and Bit shift | ~ << >> >>> & ^ \| |

**1. Simple assignment:** This is the most commonly used operator. It assigns the value on its right to the operand on its left. Here are some examples:

Assigning numbers: int x = 10; float y = 3.5F;

Assigning object references: String message = “Hello world”; File csv = new File(“test.csv”);

**2. Arithmetic operators:** The arithmetic operators are used to perform mathematic calculations just like basic mathematics in school. The following table lists all arithmetic operators in Java:

| Operator | Meaning |
|---|---|
| + | Addition (and strings concatenation) operator |
| - | Subtraction operator |
| * | Multiplication operator |
| / | Division operator |
| % | Remainder operator |

Example Program:

```
public class ArithmeticDemo
{
   public static void main(String[] args)
   {
      int x = 10;
      int y = 20;
      int result = x + y;
      System.out.println("x + y = " + result);
      result = x - y;
      System.out.println("x - y = " + result);
      result = x * y;
      System.out.println("x * y = " + result);
      result = y / x;
      System.out.println("y / x = " + result);
      result = x % 3;
      System.out.println("x % 3 = " + result);
   }
}
```

Output:

```
x + y = 30
x - y = -10
x * y = 200
y / x = 2
x % 3 = 1
```

→ The addition operator (+) can also be used for joining two or more strings together (strings concatenation). Here’s an example program:

```
public class StringConcatDemo
{
```

```
        public static void main(String[] args)
        {
                String firstName = "James";
                String lastName = "Gosling";
                String greeting = "Hello " + firstName + " " + lastName;
                System.out.println(greeting);
        }
}
```

Output:

Hello James Gosling!

**3. Unary operators:** The unary operators involve in only a single operand. The following table lists all unary operators in Java:

| Operator | Meaning |
|---|---|
| + | Unary plus operator; indicates positive value (numbers are positive by default, without this operator). |
| - | Unary minus operator; negate an expression. |
| ++ | Increment operator; increments a value by 1. |
| -- | Decrement operator; decrements a value by 1; |
| ! | Logical complement operator; inverts value of a boolean. |

Example Program:

```
public class UnaryDemo
{
   public static void main(String[] args)
   {
      int x = 10;
      int y = 20;
      int result = +x;
      System.out.println("+x = " + result);
      result = -y;
      System.out.println("-y = " + result);
      result = ++x;
      System.out.println("++x = " + result);
      result = --y;
      System.out.println("++y = " + result);
      boolean ok = false;
      System.out.println(ok);
      System.out.println(!ok);
   }
```

```
}
Output:
+x = 10
-y = -20
++x = 11
++y = 19
false
true
```

Note that the increment and decrement operators can be placed before (prefix) or after (postfix) the operand, e.g. ++x or x++, --y or y--. When using these two forms in an expression, the difference is:

Prefix form: the operand is incremented or decremented before used in the expression.

Postfix form: the operand is incremented or decremented after used in the expression.

The following example illustrates the prefix/postfix:

```
public class PrefixPostfixDemo
{
   public static void main(String[] args)
  {
      int x = 10;
      int y = 20;
      System.out.println(++x);
      System.out.println(x++);
      System.out.println(x);
      System.out.println(--y);
      System.out.println(y--);
      System.out.println(y);
   }
}
Output:
11
11
12
19
19
18
```

**4. Relational operators:** The relational operators are used to compare two operands or two expressions and result is a boolean. The following table lists all relational operators in Java.

| Operator | Meaning |
|---|---|
| == | equal to |
| != | not equal to |
| > | greater than |
| >= | greater than or equal to |
| < | less than |
| <= | less than or equal to |

```
public class RelationalDemo
{
   public static void main(String[] args)
   {
      int x = 10;
      int y = 20;
      boolean result = x == y;
      System.out.println("x == y? " + result);
      result = x != y;
      System.out.println("x != y? " + result);
      result = x > y;
      System.out.println("x > y? " + result);
      result = x >= y;
      System.out.println("x >= y? " + result);
      result = x < y;
      System.out.println("x < y? " + result);
      result = x <= y;
      System.out.println("x <= y? " + result);
   }
}
```

Output:

```
x == y? false
x != y? true
x > y? false
x >= y? false
x < y? true
x <= y? true
```

**5. Conditional operators:** The conditional operators (&& and ||) are used to perform conditional-AND and conditional-OR operations on two boolean expressions and result in a boolean value. They have "short-circuiting" behavior:

For the && operator: if the left expression is evaluated to false, then the right expression is not evaluated. Final result is false.

For the || operator: if the left expression is evaluated to true, then the right expression is not evaluated. Final result is true.

| Operator | Meaning |
|---|---|
| `&&` | conditional -AND operator |
| `\|\|` | conditional-OR operator |
| `? :` | ternary operator in form of: *A ? B : C* |

*Example Program:*

```
public class ConditionalDemo
{
  public static void main(String[] args)
  {
      int x = 10;
      int y = 20;
      if ((x > 8) && (y > 8))
      {
         System.out.println("Both x and y are greater than 8");
      }
      if ((x > 10) || (y > 10))
     {
         System.out.println("Either x or y is greater than 10");
     }
  }
}
```

Output:

```
Both x and y are greater than 8
Either x or y is greater than 10
```

Other conditional operators are ***?*** and ***:*** which form a ternary (three operands) in the following form: ***result = A ? B : C***. This is interpreted like this: if A evaluates to true, then evaluates B and assign its value to the result. Otherwise, if A evaluates to false, then evaluates C and assign its value to the result. For short, we can say: If A then B else C. So this is also referred as shorthand for an if-then-else statement.

Example:

```
public class TernaryDemo
{
   public static void main(String[] args)
  {
      int x = 10;
      int y = 20;
      int result = (x > 10) ? x : y;
      System.out.println("result 1 is: " + result);
      result = (y > 10) ? x : y;
      System.out.println("result 2 is: " + result);
   }
}
```

Output: result 1 is: 20
result 2 is: 10

**6. Type comparison operator (instanceof):** The instanceof operator tests if an object is an instance of a class, a subclass or a class that implements an interface; and result in a boolean value. Here's an example program:

```
public class InstanceofDemo
{
   public static void main(String[] args)
   {
       String name = "Java";
       if (name instanceof String)
       {
          System.out.println("an instance of String class");
       }
   }
}
```

Output:
an instance of String class

**7. Bitwise and Bit shift operators:** These operators perform bitwise and bit shift operations on only integral types, not float types. They are rarely used so the listing here is just for reference:

| Operator | Meaning |
|---|---|
| ~ | unary bitwise complement; inverts a bit pattern |
| << | signed left shift |
| >> | signed right shift |
| >>> | unsigned right shift |
| & | bitwise AND |

| | |
|---|---|
| ^ | bitwise exclusive OR |
| \| | bitwise inclusive OR |

```
public class BitDemo
{
   public static void main(String[] args)
  {
      int x = 10;
      int result = x << 2;
       System.out.println("Before left shift: " + x);
      System.out.println("After left shift: " + result);
   }
}
```

Output:
Before left shift: 10
After left shift: 40

→→ **Expressions:** An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. Java can handle any complex mathematical expressions.

| **Algebraic expression** | **Java expression** |
|---|---|
| ab-c | a*b-c |
| (m+n)(x+y) | (m+n)*(x+y) |
| $3x^2+2x+1$ | 3*x*x+2*x+1 |

Evaluation of Expression: Expressions in java are evaluated using an assignment of the form ***variable = expression;*** variable is any valid java variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted.

→→ **Precedence rules and Associativity:** Java operators have two properties those are precedence, and associativity. Precedence is the priority order of an operator, if there are two or more operators in an expression then the operator of highest priority will be executed first then higher, and then high. For example, in expression 1 + 2 * 5, multiplication (*) operator will be processed first and then addition. It's because multiplication has higher priority or precedence than addition.

If all operators in an expression have same priority in such case the second property associated with an operator comes into play, which is associativity. Associativity tells the direction of execution of operators that can be either left to right or right to left.

For example, in expression a = b = c = 8 the assignment operator is executed from right to left that means c will be assigned by 8, then b will be assigned by c, and finally a will be assigned by b. You can parenthesize this expression as (a = (b = (c = 8))).

The following is the Table which lists Java operators - precedence chart highest to lowest.

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| 1 | []<br>()<br>. | array index<br>method call<br>member access | Left -> Right |
| 2 | ++<br>--<br>+ -<br>~<br>! | pre or postfix increment<br>pre or postfix decrement<br>unary plus, minus<br>bitwise NOT<br>logical NOT | Right -> Left |
| 3 | (type cast)<br>new | type cast<br>object creation | Right -> Left |
| 4 | *<br>/<br>% | multiplication<br>division<br>modulus (remainder) | Left -> Right |
| 5 | + -<br>+ | addition, subtraction<br>string concatenation | Left -> Right |
| 6 | <<<br>>><br>>>> | left shift<br>signed right shift<br>unsigned or zero-fill right shift | Left -> Right |
| 7 | <<br><=<br>><br>>=<br>instanceof | less than<br>less than or equal to<br>greater than<br>greater than or equal to<br>reference test | Left -> Right |
| 8 | == | equal to | Left -> Right |

| | | | |
|---|---|---|---|
| | != | not equal to | |
| 9 | & | bitwise AND | Left -> Right |
| 10 | ^ | bitwise XOR | Left -> Right |
| 11 | \| | bitwise OR | Left -> Right |
| 12 | && | logical AND | Left -> Right |
| 13 | \|\| | logical OR | Left -> Right |
| 14 | ? : | conditional (ternary) | Right -> Left |
| 15 | = += -= *= /= %= &= ^= \|= <<= >>= >>>= | assignment and short hand assignment operators | Right -> Left |

→→ **Primitive Type Conversion and Casting:** Java supports two types of castings – primitive data type casting and reference type casting. Reference type casting is nothing but assigning one Java object to another object. Type casting comes with 3 categories.

**1. Implicit casting (widening conversion):** A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as automatic type conversion.

Examples:

```
int x = 10;                 // occupies 4 bytes
double y = x;               // occupies 8 bytes
System.out.println(y);      // prints 10.0
```

In the above code 4 bytes integer value is assigned to 8 bytes double value.

**2. Explicit casting (narrowing conversion):** A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires explicit casting; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

```
double x = 10.5;            // 8 bytes
int y = x;                  // 4 bytes ; raises compilation error
```

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. Let us explicitly type cast it.

```
double x = 10.5;
int y = (int) x;
```

The double x is explicitly converted to int y. The thumb rule is, on both sides, the same data type should exist.

**3. Boolean casting:** A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly; but boolean cannot. We say, boolean is incompatible for conversion. Maximum we can assign a boolean value to another boolean.

Following raises error.

```
boolean x = true;
int y = x;                // error
boolean x = true;
int y = (int) x;          // error
```

***byte –> short –> int –> long –> float –> double***

In the above statement, left to right can be assigned implicitly and right to left requires explicit casting. That is, byte can be assigned to short implicitly but short to byte requires explicit casting.

Example:

```
public class Demo
{
 public static void main(String args[])
 {
   char ch1 = 'A';
   double d1 = ch1;

   System.out.println(d1);             // prints 65.0
   System.out.println(ch1 * ch1);      // prints 4225 , 65 * 65

   double d2 = 66.0;
   char ch2 = (char) d2;
   System.out.println(ch2);            // prints B
 }
}
```

→→**Flow of Control:** A Java program is a set of statements, which are normally executed sequentially in the order in which they appear. This happens when options or repetitions of certain calculations are not necessary.

When a program breaks the sequential flow and jumps to another part of the code, it is called *branching*. When the branching is based on a particular condition, it is known as *conditional branching*. It the branching takes place without any decision, it is known as *unconditional branching*.

**1. Conditional Statements:** these conditional statements include the following

*a) if else statement*

*b) nested if statement*

*c) if else if ladder*

*d) switch case statement*

**2. Iteration Statements:** these iteration statements include the following

*a) while loop*

*b) do while loop*

*c) for loop*

**3. Jump Statements:** these jump statements include the following

*a) break*

*b) continue*

*c) return*

**1. Conditional Statements:**

**a) if else statement:** This statement is used to perform a task depending upon whether the given condition is true or false. Here a task represents single statement or group of statements.

**Syntax:** if(condition)

```
{
//statements;
}
else
{
//Statements;
}
```

Here if condition is true then statement1 will be executed. if condition is false then statement2 will be executed. statement1 and statement2 represent either a single statements or more than one statement. If more than one statement is used then they should be enclosed in angular bracket {}.

**E.g.:**
```
if(a>b)
    System.out.println("the value of a=" +a);
else
    System.out.println("the value of b=" +b);
```

**b) Nested if statement:** A nested **if** is an **if** statement that is the target of another **if** or **else.** Nested **if**s are very common in programming. When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

**Syntax:**
```
if(condition1)
    {
      if(condition2)
        //Statements1;
      else
        //Statements2;
    }
    else
        //Statements3;
```

**E.g.:**
```
if(i == 10)
  {
    if(j < 20)
      a = b;
    else
      a = c; // associated with this else
  }
  else
      a = d; // this else refers to if(i == 10)
```

**c) if-else-if Ladder:** A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if ladder*.

**Syntax:**
```
if(condition)
  // statement;
else if(condition)
      //  statement;
```

```
else if(condition)
 // statement;
else
  //statement;
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

**E.g.:**

```
int fib(int m)
    {
       if(m==1)
              return 1;
       else
      if(m==2)
                return 1;
      else
                return (fib(m-1)+fib(m-2));
    }
```

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

**Syntax:**

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

→The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression.

→Each **case** value must be a unique literal (that is, it must be a constant, not a variable).

→Duplicate **case** values are not allowed.

**E.g.:**
```
switch(i)
   {
     case 0: System.out.println("i is zero.");
              break;
     case 1: System.out.println("i is one.");
              break;
     case 2: System.out.println("i is two.");
              break;
     case 3: System.out.println("i is three.");
             break;
     default: System.out.println("i is greater than 3.");
  }
```

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

**a) while loop:** The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.

**Syntax:**
```
while(condition)
{
// body of loop
}
```

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

**E.g.:**
```
while(c<n)
    {
       a=b;
       b=c;
       c=a+b;
    }
```

**b) do-while loop:** If the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

**Syntax:** do
```
{
        // body of loop
} while (condition);
```
→Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

→The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once. Consider the following program, which implements a very simple help system for Java's selection and iteration statements:

**E.g.:**
```
int  n=10;
 do
{
   System.out.println(n);
    n--;
 }while(n>0);
```

**c) for loop:** The traditional form that has been in use since the original version of Java. When the loop starts, the initialization portion of the loop is executed. Initialization expression executed only once and then condition is evaluated. This must be a Boolean expression. Check the condition; if the condition is true, then the body of the loop is executed. If it is fail, the loop terminates. Next, the iteration portion of the loop is executed and goes to increment or decrement the loop control variable and again checks the condition if it is true, then the body of the loop is executed. This process is repeated up to the condition is fail.

**Syntax**: traditional **for** statement**:**
```
for(initialization; condition; iteration)
{
```

```
// body
}
```
**E.g.:**
```
for(i=1;i<=n;i++)
    {
        a=b;
        b=c;
        c=a+b;
    }
```
**Declaring Loop Control Variables Inside the for Loop**
```
for(datatype initialization; condition; iteration)
{
    // body

}
```
**E.g.:**
```
for(int i=1;i<=n;i++)
    {
        a=b;
        b=c;
        c=a+b;
```
**3. Jump Statements:** Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

**a) break:** In Java, the **break** statement has three uses.

- it terminates a statement sequence in a **switch** statement.
- it can be used to exit a loop.
- it can be used as a "civilized" form of goto.

→ The last two uses are explained here.

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

**E.g.:**
```
for(int i=0; i<100; i++)
  {
    if(i == 10)
   break; // terminate loop if i is 10
   System.out.println("i: " + i);
  }
```
***Labelled break:*** java supports labelled break

**Syntax:** break *label*;
```
boolean t = true;
```

```
first:
{
second:
{
third:
{
System.out.println("Before the break.");
if(t)
 break second; // break out of second block
System.out.println("This won't execute");
}
}
System.out.println("This is after second block.");
}
```

**Output:**

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

**b) continue:** The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

**E.g.:**

```
for(int i=0; i<10; i++)
    {
        System.out.print(i + " ");
        if (i%2 == 0) continue;
       System.out.println("");
    }
```

**c) return:** The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. At any time in a method the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main( )**.

**E.g.:**

```
int fib(int m)
    {
        if(m==1)
        return 1;
```

```
        else if(m==2)
         return 1;
        else
        return (fib(m-1)+fib(m-2));
    }
```

→→ **Classes and Objects:** Java is a true object-oriented language and therefore a java program must be encapsulated in a class that defines the state and behavior of the basic program components known as Objects.

Class: A class can be defined as a template/blue print that describes the behaviors/states that object of its type support. It is a combination of data items and functions which are called as members of class.

**The General Form of a Class**

```
class ClassName [extends SuperClassName]
{
        [type instanceVariable1;]
        [type instanceVariable2;]
        [type methodName1(parameter-list)
        {
                // body of method
        }]
        [type methodName2(parameter-list)
         {
                // body of method
        }]
}
```

Everything inside the square brackets is optional. Therefore, the following is a valid class definition:

```
class Empty
{
}
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called

members of the class. Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.

**Object:** An object in java is essentially a block of memory that contains space to store the instance variables. Creating an object is also refers to as instantiating an object. Objects are created using ***"new"*** operator. The new operator dynamically allocates memory for an object and returns reference to that object.

```
ClassName cn;          //declare the reference variable
cn=new ClassName(); //instantiate the object
```

The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable. The variable cn is now an object of the ClassName. Both Statements can be combined into one statement: ClassName cn=new ClassName();

It is important to understand that any changes to the variables of one object have no effect on the variables of another.

**Accessing Class Members:** Each object of a class is having its own set of variables, we should assign values to these variables in order to use them in our program. The following is the procedure for accessing a variable and a method respectively.

*objectname.variablename = value;*

*objectname.methodname();*

Example: Program to illustrate class members:

```
import java.util.*;
class Example
{
        int num1,num2;
        void method()
        {
                System.out.println("Num1 value is "+num1+" Num2 value is "+num2);
        }
        public static void main(String args[])
        {
                Example e=new Example();
                e.num1=12;
                e.num2=22;
                e.method();
        }
}
```

Output:
Num1 value is 12 Num2 value is 22

**Methods:** Classes usually consist of two things: instance variables and methods. Method is a large one because Java gives them so much power and flexibility. **This is the general form of a method:**

*return type methodName*(*parameter-list*)
{
// body of method
}
Here, return *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**.

→Method name can be any legal identifier other than those already used by other items within the current scope.

→The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called.

→If the method has no parameters, then the parameter list will be empty.

→Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement: return *value*; here, *value* is the value returned.

Example:

```
class MethodDemo
{
        int a,b,c;
        int show()
        {
                return a*b*c;
        }
        void var(int x,int y, int z)
        {
                a=x;
                b=y;
                c=z;
        }
}
class ParaMain
```

```
{
        public static void main(String args[])
        {
                int d;
                MethodDemo mdr=new MethodDemo();
                md.var(10,2,3);
                d=md.show();
                System.out.println("D value: "+d);
        }
}
```

Output:
D value: 60

→→ **Constructors:** Constructor in java is a special type of method that is used to initialize the object. Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

There are basically two rules defined for the constructor.

- Constructor name must be same as its class name
- Constructor must have no explicit return type

There are two types of constructors:

- Default constructor (no-arg constructor): A constructor that has no parameter is known as default constructor.
- Parameterized constructor: A constructor that has parameters is known as parameterized constructor.

*Default constructor*: Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example:

```
class ConstructorDemo
{
        ConstructorDemo()
        {
                System.out.println(“Default Constructor”);
        }
        public static void main(String args[])
        {
                ConstructorDemo cd=new ConstructorDemo();
        }
```

```
}
```
Output: Default Constructor

Note: If there is no constructor in a class, compiler automatically creates a default constructor.

*Parameterized constructor:* Parameterized constructor is used to provide different values to the distinct objects.

Example:

```
class ConstructorDemoOne
{
        int num1,num2;
        ConstructorDemoOne(int n1,int n2)
        {
                num1=n1;
                num2=n2;
        }
        void display()
        {
                System.out.println("Num1 value is "+num1+" Num2 value is "+num2);
        }
        public static void main(String args[])
        {
                ConstructorDemoOne cd=new ConstructorDemoOne(2,5);
                cd.display();
        }
}
```
Output:
Num1 value is 2 Num2 value is 5.

→ **Constructor Overloading:** A constructor can also be overloaded. Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters. Constructor overloading is not much different than method overloading. In case of method overloading you have multiple methods with same name but different signature, whereas in Constructor overloading you have multiple constructor with different signature but only difference is that Constructor doesn't have return type in Java. Constructor overloading is done to construct object in different ways.

Example:

```
class ConstructorOLDemo
{
        int num1,num2;
        ConstructorOLDemo()
        {
                num1=num2=0;
        }
        ConstructorOLDemo(int n1,int n2)
```

```
        {
                num1=n1;
                num2=n2;
        }
        void display()
        {
                System.out.println("Num1
value is "+num1+" Num2 value is "+num2);
        }
        public static void main(String args[])
        {
                ConstructorOLDemo cd=new
ConstructorOLDemo();
                cd.display();
                ConstructorOLDemo
cd1=new ConstructorOLDemo(2,5);
                cd1.display();
        }
}
```

Output:
Num1 value is 0 Num2 value is 0.
Num1 value is 2 Num2 value is 5.

→ There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

```
class CopyDemo
{
        int num1,num2;
        CopyDemo ()
        {
                num1=num2=0;
        }
        CopyDemo (int n1,int n2)
        {
                num1=n1;
                num2=n2;
        }
        CopyDemo(CopyDemo cd)
        {
                num1=cd.num1;
                num2=cd.num2;
        }
        void display()
        {
                System.out.println("Num1 value is "+num1+" Num2 value is "+num2);
        }
        public static void main(String args[])
        {
                CopyDemo cd=new CopyDemo();
                cd.display();
                CopyDemo cd1=new CopyDemo(2,5);
                cd1.display();
```

```
            CopyDemo cd2=new CopyDemo(cd1);
            cd2.display();

        }
}
```

Output:
Num1 value is 0 Num2 value is 0.
Num1 value is 2 Num2 value is 5.
Num1 value is 2 Num2 value is 5.

→ **Garbage Collector:** In java, garbage means unreferenced objects. Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using *free()* function in C language and *delete()* in C++. But, in java it is performed automatically. So, java provides better memory management.

*Advantage of Garbage Collection*: It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory. It is automatically done by the garbage collector (a part of JVM) so we don't need to make extra efforts.

An object can be unreferenced. There are many ways:

- By nulling the reference
- By assigning a reference to another

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

→ *finalize() method:* The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as: *protected void finalize(){}*

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

→ *gc() method*: The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes. *public static void gc(){}.*

*Note: Neither finalization nor garbage collection is guaranteed.*

Example:

```
public class TestGarbage1
{
        public void finalize()
        {
                System.out.println("object is garbage collected");
        }
        public static void main(String args[])
        {
                TestGarbage1 s1=new TestGarbage1();
                TestGarbage1 s2=new TestGarbage1();
                s1=null;
                s2=null;
                System.gc();
        }
}
```

Output:

```
object is garbage collected
object is garbage collected
```

→→ **this keyword:** this is a reference variable that refers to the current object. The following are the usages of this keyword.

- this keyword can be used to refer current class instance variable.
- this() can be used to invoke current class constructor.
- this keyword can be used to invoke current class method (implicitly)
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this keyword can also be used to return the current class instance.

Example for this keyword can be used to refer current class instance variable.

```
class Box
{
        double width,height,length;
        Box()
        {
                width=length=height=-1;
        }
```

```
        Box(double width, double height, double length)
        {
                this.width=width;
                this.height=height;
                this.length=length;
        }
        double volume()
        {
                return width*length*height;
        }
        public static void main(String args[])
        {
                Box b=new Box();
                double vol=b.volume();
                System.out.println("Volume of Box is "+vol);
                Box b1=new Box(12,12,12);
                b1.volume();
                System.out.println("Volume of Box is "+vol);
        }
}
```

Output:
Volume of Box is -1
Volume of Box is 1728

Example for this() can be used to invoke current class constructor.

```
class ThisDemo
{
        int num1,num2;
        ThisDemo(int num1,int num2)
        {
                this.num1=num1;
                this.num2=num2;
        }
        ThisDemo(int num1)
        {
                this(num1,num1);
        }
        ThisDemo()
        {
                this(0);
        }
        public static void main(String args[])
        {
                ThisDemo td=new ThisDemo();
        }
}
```

Explanation: this() is used for calling the current class constructor. this(0) calls the constructor which is having a single parameter constructor of type integer. this(num1,num1) calls the constructor which is having two parameters of type integer.

→→ **Static Keyword:** If we want to define a class member where that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. But, it is possible to create a member that can be used by

itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static.

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.

The members that are declared as static are called static members. Since these members are associated with the class itself rather than individual objects, the static variables and static methods are referred as class variables and class methods in order to distinguish them from instance variables and instance methods.

Static variables are used when we want to have a variable common to all instances of a class. Java creates only one copy for a static variable which can be used even if the class in not instantiated.

Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to this or super in any way.

We can declare static block that gets executed exactly once, when the class is first loaded. static variables and static methods can be used independently of any object. For this, we need to specify the name of their class followed by the dot operator.

Syntax: ClassName.staticVariable;
ClassName.staticMethod();

Example: // Demonstrate static variables, methods, and blocks.

```
class UseStatic
{
   static int a = 3;
   static int b;
   static void meth(int x)
   {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
   }
   static
   {
System.out.println("Static block initialized.");

        b = a * 4;
        }
}
class StaticDemo
{
        public static void main(String args[])
```

```
        {
                UseStatic.meth(42);
        }
}
Output:
```

```
Static block initialized.
x = 42
a = 3
b = 12
```

→→ **Arrays:** An array is a group of variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index.

→ **One-Dimensional Arrays:** A one-dimensional array is a list of variables.

- To create an array, first you must create an array variable of the desired type.

*Syntax: type[] var-name;*

- type declares the base type of the array.
- The base type determines the data type of each element that comprises the `array. Thus, the base type for the array determines what type of data the array will hold.

- new is a special operator for allocating memory.

*Syntax var-name = new type[size];*

- type specifies the type of data being allocated.
- size specifies the number of elements in the array which must be a numeric constant.
- var-name is the array variable that is linked to the array.

The elements in the array allocated by new will automatically be initialized to zero.

E.g.*: int[] numbers = new int[3];*

An array is a two-step process:

- You must declare a variable of the desired array type.
- You must allocate the memory that will hold the array, using new, and assign it to the array variable.

Thus, in Java all arrays are dynamically allocated. Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.

// Demonstrate a one-dimensional array.

```
class Array
{
        public static void main(String args[])
        {
                int numbers[];
                numbers = new int[3];
                numbers [0] = 4;
                numbers [1] = 3;
                numbers [2] = 6;
                System.out.println("I Like Number "+ numbers [0]);
        }
}
```

Output:
I Like Number 4

- An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use new.

```
// An improved version of the previous program.
class AutoArray
{
public static void main(String args[])
 {
int[] numbers = { 4,3,6 };
System.out.println("I Like Number "+ numbers [0] );
}
}
```

Output:
I Like Number 4

→ **Alternative Array Declaration Syntax:** There is a second form that may be used to declare an array: *Syntax: type var-name[];*

Here, the square brackets follow the array variable name, and not the type specifier.

E.g.: the following two declarations are equivalent:

*int[] a2 = new int[3];*

*int al[] = new int[3];*

The following declarations are also equivalent:

*char[][] twod2 = new char[3][4];*

*char twod1[][] = new char[3][4];*

This first declaration form offers convenience when declaring several arrays at the same time.

*E.g. int[] nums, nums2, nums3; // create three arrays*

The above statement creates three array variables of type int. It is the writing same as the following: *int nums[], nums2[], nums3[]; // create three arrays*

The first declaration form is also useful when specifying an array as a return type for a method.

→ **Multidimensional Arrays:** In Java, multidimensional arrays are actually arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

E.g.*: int[][] TD = new int[3][3];*

This allocates a 3 by 3 array and assigns it to TD. Internally this matrix is implemented as an array of arrays of int.

The following program numbers each element in the array, row by row, and then displays these values:

```
// Demonstrate a two-dimensional array.
class TwoDArray
{
        public static void main(String args[])
        {
                int i ,j ,k=0;
                int TD[][]= new int[3][3];
                for(i=0; i<3; i++)
                {
                        for( j=0; j<3; j++)
                        {
                                k=TD[i][j];
                                k++;
                        }
                }
                for(i=0; i<3; i++)
                {
                for(j=0; j<3; j++)
                {
                System.out.print(TD[i][j] + " ");
                }
                System.out.println();
                }
                }
                }
                Output:
                0 1 2
                3 4 5
                6 7 8
```

→→ **foreach loop:** JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

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

```
                double[] myList = {1,2,3,4};
                for(double ele:myList)
                {
                        System.out.print(ele);
                }
}       }
```

Output: E:\> javac TestArray.java
E:\ >java TestArray
1.0 2.0 3.0 4.0

→→ **Command Line Arguments:** String[] args within the declaration of the main method represents a String array named args where args is nothing more than an identifier, we can replace it with any other identifier and the program will still work. A String array can be passed as an argument when we execute the program. We pass it through the command line itself. Consider that we have a class named Add. The following statement normally used to execute the program.

E:\>javac Add.java

E:\>java Add

When we wish to pass the String array, we simply include the elements of the array as simple Strings beside the class name. Enclosing the Strings in quotes is optional. Consecutive Strings are separated with a space. For example, if we wish to pass a three element String array containing the values "1", "2", and "3" any of the following lines is entered on the command prompt.

E:\>java Add 1 2 3

E:\>java Add "1" "2" "3"

Since these arguments are passed through the command line, they are known as command line arguments. The String arguments passed are stored in the array specified in the main() declaration. args[] is now a three element String array. These elements are accessed in the same way as the elements of a normal array. The following is the complete Add program which is capable of adding any number of integers passed as command line arguments.

```
public class Add
{
        public static void main(String[] args)
        {
                int sum = 0;
                for (int i = 0; i < args.length; i++)
```

```
                {
                        sum = sum + Integer.parseInt(args[i]);
                }
                System.out.println("The sum of the arguments passed is " + sum);
        }
}
```

Output:

```
E:\ >java Add 1 2 3
The sum of the arguments passed is 6
```