JAVA Programming

UNIT-1

Java Introduction

- Java is a popular programming language, created in 1995.
- It is owned by Oracle, and more than **3 billion** devices run Java.
- **It is used for:**
    - Mobile applications (specially Android apps)
    - Desktop applications
    - Web applications
    - Web servers and application servers
    - Games
    - Database connection
    - And much, much more!

Why Use Java?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming language in the world
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has a huge community support (tens of millions of developers)
- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
- As Java is close to C++ and C#, it makes it easy for programmers to switch to Java or vice versa

Programming language

A **programming language** is a formal language that specifies a set of instructions that can be used to produce various kinds of output. Programming languages generally consist ofinstructions for a computer. Programming languages can be used to create programs that implement specific algorithms.

Types of Computer Language

- **Machine Language:** a language that is directly interpreted into the hardware
- **Assembly Language:** a slightly more user-friendly language that directly corresponds to machine language

- **High-level language** :High-level languages are basically symbolic languages that use English words and/or mathematical symbols rather than mnemonic codes. Each instruction in the high-level language is translated into many machine language instructions that the computer can understand.

Machine languages

Machine language is the lowest and most elementary level of programming language and was the first type of programming language to be developed. Machine language is basically the only language that a computer can understand and it is usually written in hex.In fact, a manufacturer designs a computer to obey just one language, its machine code, which is represented inside the computer by a string of binary digits (bits) 0 and 1. The symbol 0 stands for the absence of an electric pulse and the 1 stands for the presence of an electric pulse. Since a computer is capable of recognizing electric signals, it understands machine language.

Advantages:
Fast execution 2.It requires no translator to translate the code. It is directly understood by the computer
Disadvantages:
1. Programs had to be written using binary codes unique to each computer.
2. Programmers had to have a detailed knowledge of the internal operations of the specific type of CPU they were using.
3. Programming was difficult and error-prone
4. Programs are not portable to other computers.

Assembly language

Assembly language was developed to overcome some of the many inconveniences of machine language. This is another low-level but very important language in which operation codes and operands are given in the form of alphanumeric symbols instead of 0's and l's.

These alphanumeric symbols are known as mnemonic codes and can combine in a maximum of five-letter combinations e.g. ADD for addition, SUB for subtraction, START, LABEL etc. Because of this feature, assembly language is also known as 'Symbolic Programming Language.'

Advantages:
1. Uses symbolic coded instructions which are easier to remember
2. Programming is simplified as a programmer does not need to know the exact storage location of data and instructions.
3. Efficient use of computer resources is out weighted by the high costs of very tedious systems development and by lock of program portability.

Disadvantage:
1. Assembler languages are unique to specific types of computers.
2. Programs are not portable to other computers.

High-Level Languages (procedural)
High-level computer languages use formats that are similar to English. The purpose of developing high-level languages was to enable people to write programs easily, in their own native language environment (English).
High-level languages are basically symbolic languages that use English words and/or mathematical symbols rather than mnemonic codes. Each instruction in the high-level language is translated into many machine language instructions that the computer can understand.
Advantages:
1. Easier to learn and understand than an assembler language as instructions (statements) that resemble human language or the standard notation of mathematics.
2. Have less-rigid rules, forms, and syntaxes, so the potential for error is reduced.
3. Are machine-independent programs therefore programs written in a high-level language do not have to be reprogrammed when a new computer is installed.
4. Programmers do not have to learn a new language for each computer they program.
Disadvantages:
1. Less efficient than assembler language programs and require a greater amount of computer time for translation into machine instructions.
Types of High-Level Languages
Many languages have been developed for achieving a variety of different tasks. Some are fairly specialized, and others are quite general.
These languages, categorized according to their use, are:
1) Algebraic Formula-Type Processing
These languages are oriented towards the computational procedures for solving mathematical and statistical problems.
Examples include:
- BASIC (Beginners All Purpose Symbolic Instruction Code)
- FORTRAN (Formula Translation)
- PL/I (Programming Language, Version 1)
- ALGOL (Algorithmic Language)
- APL (A Programming Language)

2. Business Data Processing
These languages are best able to maintain data processing procedures and problems involved in handling files. Some examples include:
- COBOL (Common Business Oriented Language)
- RPG (Report Program Generator)

3. String and List Processing

These are used for string manipulation, including search patterns and inserting and deleting characters. Examples are:

- LISP (List Processing)
- Prolog (Program in Logic)

4. Object-Oriented Programming Language

In OOP, the computer program is divided into objects. Examples are:

- C++
- Java

5. Visual Programming Language

These programming languages are designed for building Windows-based applications.Examples are:

- Visual Basic
- Visual Java
- Visual C

Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java *buzzwords*.

A list of most important features of Java language is given below.

- Simple
- Object-Oriented
- Portable
- Platform independent
- Secured
- Robust
- Architecture neutral
- Interpreted
- High Performance
- Multithreaded
- Distributed
- Dynamic

Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).

- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object-oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

1. **Basic concepts of OOPs are:**
2. **Object**
3. **Class**
4. **Inheritance**
5. **Polymorphism**
6. **Abstraction**
7. **Encapsulation**

Platform Independent

Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

- Runtime Environment
- API(Application Programming Interface)

Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**
- **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by

separating the package for the classes of the local file system from those that are imported from network sources.

- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

Robust

- Robust simply means strong. Java is robust because:
- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-

threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

Java Example Program

Creating a Hello World Program in Java is not a single line program. It consists of various other lines of code. Since Java is a Object-oriented language so it require to write a code inside a class. Let us look at a simple java program.

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

keypoints.

- class : class keyword is used to declare classes in Java
- public : It is an access specifier. Public means this function is visible to all.
- static : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The main() method here is called by JVM, without creating any object for class.
- void : It is the return type, meaning this function will not return anything.
- main : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.
- String[] args : This represents an array whose type is String and name is args. We will discuss more about array in Java Array section.
- System.out.println : This is used to print anything on the console like *printf* in C language.

Steps to Compile and Run your first Java program

Step 1: Open a text editor and write the code as above.

Step 2: Save the file as Hello.java

Step 3: Open command prompt and go to the directory where you saved your first java program assuming it is saved in C drive.

Step 4: Type javac Hello.java and press Return(Enter KEY) to compile your code. This command will call the Java Compiler asking it to compile the specified file. If there are no errors in the code the command prompt will take you to the next line.

Step 5: Now type java Hello on command prompt to run your program.
Step 6: You will be able to see Hello world program printed on your command prompt.

C++ vs Java

C++	Java
C++ is platform-dependent.	Java is platform-independent.
C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
C++ was designed for systems and applications programming. It was an extension of C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed with a goal of being easy to use and accessible to a broader audience.
C++ supports the goto statement.	Java doesn't support the goto statement.
C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
C++ supports operator overloading.	Java doesn't support operator overloading.
C++ supports pointers. You can write pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses compiler and interpreter both. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform independent.
C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
C++ supports structures and unions.	Java doesn't support structures and unions.

C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
C++ doesn't support documentation comment.	Java supports documentation comment (/** ... */) to create documentation for java source code.
C++ supports virtual keyword so that we can decide whether or not override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
C++ creates a new inheritance tree always.	Java uses a single inheritance tree always because all classes are the child of Object class in java. The object class is the root of the inheritance tree in java.
C++ is nearer to hardware.	Java is not so interactive with hardware.
C++ is an object-oriented language. However, in C language, single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.

JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

Object Oriented Programming (OOPs) Concept in Java

Object-oriented programming: As the name suggests, Object-Oriented Programming or OOPs refers to languages that uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

OOPs Concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Class
- Object
- Method
- Message Passing

1. **Polymorphism:** Polymorphism refers to the ability of OOPs programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities.

For example:

```
// Java program to demonstrate Polymorphism
  // This class will contain
// 3 methods with same name,
// yet the program will
// compile & run successfully
public class Sum {
     // Overloaded sum().
   // This sum takes two int parameters
   public int sum(int x, int y)
   {
      return (x + y);
   }
    // Overloaded sum().
   // This sum takes three int parameters
   public int sum(int x, int y, int z)
   {
      return (x + y + z);
   }
```

```
  // Overloaded sum().
 // This sum takes two double parameters
 public double sum(double x, double y)
 {
    return (x + y);
 }
  // Driver code
 public static void main(String args[])
 {
    Sum s = new Sum();
    System.out.println(s.sum(10, 20));
    System.out.println(s.sum(10, 20, 30));
    System.out.println(s.sum(10.5, 20.5));
 } }
```

Output:

30

60

31.0

Polymorphism in Java are mainly of 2 types:

- Overloading in Java
- Overriding in Java

2.Inheritance: Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

Important terminology:

Super Class: The class whose features are inherited is known as superclass(or a base class or a parent class).

Sub Class: The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

The keyword used for inheritance is **extends**.

Syntax:

```
class derived-class extends base-class
{
  //methods and fields
```

}

3.Encapsulation:

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.
Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

4.Abstraction:

Data Abstraction is the property by virtue of which only the essential details are displayed to the user.The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.
Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

5.Class:

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

- **Modifiers**: A class can be public or has default access (Refer this for details).
- **Class name:** The name should begin with a initial letter (capitalized by convention).
- **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body surrounded by braces, { }.

6.Object:

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

- **State** : It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior** : It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog

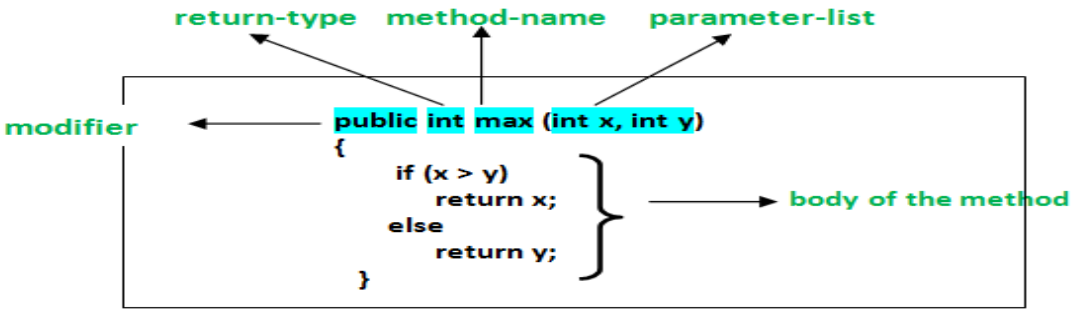
7.**Method:** A method is a collection of statements that perform some specific task and return result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++ and Python.

Method Declaration

In general, method declarations has six components:

Access Modifier: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.

- **public:** accessible in all class in your application.
- **protected:** accessible within the package in which it is defined and in its **subclass(es)(including subclasses declared outside the package)**
- **private:** accessible only within the class in which it is defined.
- **default (declared/defined without using any modifier):** accessible within same class and package within which its class is defined.
- **The return type**: The data type of the value returned by the method or void if does not return a value.
- **Method Name**: the rules for field names apply to method names as well, but the convention is a little different.
- **Parameter list**: Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
- **Exception list**: The exceptions you expect by the method can throw, you can specify these exception(s).
- **Method body**: it is enclosed between braces. The code you need to be executed to perform your intended operations.



8.**Message Passing:**

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Java Variables

- A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.
- Variable is a name of memory location.
- There are three types of variables in java:
  - ✓ local,
  - ✓ instance
  - ✓ static.

Variable

Variable is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*. It is a combination of "vary + able" that means its value can be changed.

int data=50;//Here data is variable

Types of Variables

There are three types of variables in Java:

1. **local variable**
2. **instance variable**
3. **static variable**

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static.It is called instance variable because its value is instance specific and is not shared among instances.

3) Static variable

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
class A{
int data=50;//instance variable
```

```
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

Java Variable Example: Add Two Numbers

```
class Simple{
public static void main(String[] args){
int a=10;
int b=10;
int c=a+b;
System.out.println(c);
}}
```

Output:

20

constant in Java

- A constant is a variable whose value **cannot change once it has been assigned**. Java doesn't have built-in support for constants.
- A constant can make our program more easily read and understood by others. In addition, a constant is cached by the JVM as well as our application, so using a constant can improve performance.
- To define a variable as a constant, we just need to add the keyword "**final**" in front of the variable declaration.

Syntax

```
final float pi = 3.14f;
```

The above statement declares the float variable "pi" as a constant with a value of 3.14f. We cannot change the value of "pi" at any point in time in the program.

Example

```
public class ConstantsDemo {
  public static void main(String args[]) {
    final byte var1 = 2;
    final byte var2;
    var2 = -3;
    final short var3 = 32;
    final short var4;
    var4 = -22;
    final int var5 = 100;
    final int var6;
    var6 = -112;
    final long var7 = 20000;
```

```
    final long var8;
    var8 = -11223;
    final float var9 = 21.23f;
    final float var10;
    var10 = -121.23f;
    final double var11 = 20000.3223;
    final double var12;
    var12 = -11223.222;
    final boolean var13 = true;
    final boolean var14;
    var14 = false;
    final char var15 = 'e';
    final char var16;
    var16 = 't';
   // Displaying values of all variables
    System.out.println("value of var1 : "+var1);
    System.out.println("value of var2 : "+var2);
    System.out.println("value of var3 : "+var3);
    System.out.println("value of var4 : "+var4);
    System.out.println("value of var5 : "+var5);
    System.out.println("value of var6 : "+var6);
    System.out.println("value of var7 : "+var7);
    System.out.println("value of var8 : "+var8);
    System.out.println("value of var9 : "+var9);
    System.out.println("value of var10 : "+var10);
    System.out.println("value of var11 : "+var11);
    System.out.println("value of var12 : "+var12);
    System.out.println("value of var13 : "+var13);
    System.out.println("value of var14 : "+var14);
    System.out.println("value of var15 : "+var15);
    System.out.println("value of var16 : "+var16);
  }}
```

Output

```
value of var1 : 2
value of var2 : -3
value of var3 : 32
value of var4 : -22
value of var5 : 100
value of var6 : -112
```

Java Literals

- A literal is a source code representation of a fixed value.
- They are represented directly in the code without any computation.
- Literals can be assigned to any primitive type variable.

For example −

```
byte a = 68;
char a = 'A';
```

- byte, int, long, and short can be expressed in decimal(base 10), 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.

For example −

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

String literals in Java are specified like 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"
- "two\nlines"
- "\"This is in quotes\""

String and char types of literals can contain any Unicode characters. For **example −**

```
char a = '\u0001';
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are −

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab

\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

1.Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

1. boolean data type
2. byte data type
3. char data type
4. short data type
5. int data type
6. long data type
7. float data type
8. double data type

Data Type	Default Value	**Default size**
Boolean	false	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte

Short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte
Double	0.0d	8 byte

1.Boolean Data Type

- The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.
    - The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.**Example:** Boolean one = false

2.Byte Data Type

- The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.
- The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.
  Example: byte a = 10, byte b = -20

3.Short Data Type

- The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.
- The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.
  Example: short s = 10000, short r = -5000

4.Int Data Type

- The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.
- The int data type is generally used as a default data type for integral values unless if there is no problem about memory.
  Example: int a = 100000, int b = -200000

5.Long Data Type

- The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.
  Example: long a = 100000L, long b = -200000L

6.Float Data Type

- The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.
  Example: float f1 = 234.5f

7.Double Data Type

- The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.
  Example: double d1 = 12.3

8.Char Data Type

- The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.
  Example: char letterA = 'A'

Java Strings

- Strings are used for storing text.
- A String variable contains a collection of characters surrounded by double quotes:
  Example
  Create a variable of type String and assign it a value:
  String greeting = "Hello";

String Length

A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the length() method:

Example

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println("The length of the txt string is: " + txt.length());
```

More String Methods

There are many string methods available, for example toUpperCase() and toLowerCase():

Example

```
String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   // Outputs "hello world"
```

Finding a Character in a String

The indexOf() method returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace):

Example

```
String txt = "Please locate where 'locate' occurs!";
System.out.println(txt.indexOf("locate")); // Outputs 7
```

Java counts positions from zero.

0 is the first position in a string, 1 is the second, 2 is the third ...

String Concatenation

The + operator can be used between strings to combine them. This is called **concatenation**:

Example

```
String firstName = "John";
System.out.println(firstName + " " + lastName);
```

Note that we have added an empty text (" ") to create a space between firstName and lastName on print.

You can also use the concat() method to concatenate two strings:

Example

```
String firstName = "John ";
String lastName = "Doe";
System.out.println(firstName.concat(lastName));
```

Special Characters

Because strings must be written within quotes, Java will misunderstand this string, and generate an error:

```
String txt = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

The sequence \" inserts a double quote in a string:
Example
String txt = "We are the so-called \"Vikings\" from the north.";

The sequence \' inserts a single quote in a string:
Example
String txt = "It\'s alright.";

The sequence \\ inserts a single backslash in a string:
Example
String txt = "The character \\ is called backslash.";
Six other escape sequences are valid in Java:

Code	Result	Try it
\n	New Line	Try it »
\r	Carriage Return	Try it »
\t	Tab	Try it »
\b	Backspace	Try it »
\f	Form Feed	

Adding Numbers and Strings
WARNING!

- Java uses the + operator for both addition and concatenation.
- Numbers are added. Strings are concatenated.
- If you add two numbers, the result will be a number:

Example
int x = 10;

```
int y = 20;
int z = x + y;      // z will be 30 (an integer/number)
```

Operators in Java

Operator in Java is a symbol which is used to perform operations.
For example: +, -, *, / etc.
Types of operators

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Example1: ++ and --

```
class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Arithmetic Operators

- Java arithmatic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output:

```
15
5
50
2
0
```

Left Shift Operator

- The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Example

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}}
```

Output:

```
40
80
80
240
```

Right Shift Operator

- The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.

Example

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
```

```
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Output:

```
2
5
2
```

Shift Operator Example: >> vs >>>

```
class OperatorExample{
public static void main(String args[]){
   //For positive number, >> and >>> works same
   System.out.println(20>>2);
   System.out.println(20>>>2);
   //For negative number, >>> changes parity bit (MSB) to 0
   System.out.println(-20>>2);
   System.out.println(-20>>>2);
}}
```

Output:

```
5
5
-5
1073741819
```

AND Operator : Logical && and Bitwise &

- The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.
- The bitwise & operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

Output:

```
false
false
```

AND Operator Example: Logical && vs Bitwise &

```
class OperatorExample{
```

```
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
false
10
false
11
```

OR Operator : Logical || and Bitwise |

- The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.
- The bitwise | operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
true
true
true
10
true
11
```

Ternary Operator

- Java Ternary operator is used as one liner replacement for if-then-else statement and used a lot in Java programming. it is the only conditional operator which takes three operands.

Example

```
class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

2

Assignment Operator

Java assignment operator is one of the most common operator. It is used to assign the value on its right to the operand on its left.

Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}
}
```

Output:

14
16

constant

- A constant is a variable whose value **cannot change once it has been assigned**. Java doesn't have built-in support for constants.
- A constant can make our program more easily read and understood by others. In addition, a constant is cached by the JVM as well as our application, so using a constant can improve performance.
- To define a variable as a constant, we just need to add the keyword "**final**" in front of the variable declaration.

Syntax

final float pi = 3.14f;

- The above statement declares the float variable “pi” as a constant with a value of 3.14f. We cannot change the value of "pi" at any point in time in the program. Later if we try to do that by using a statement like “pi=5.25f”, Java will throw errors at compile time itself. It is not mandatory that we need to assign values of constants during initialization itself.

Type Casting in Java

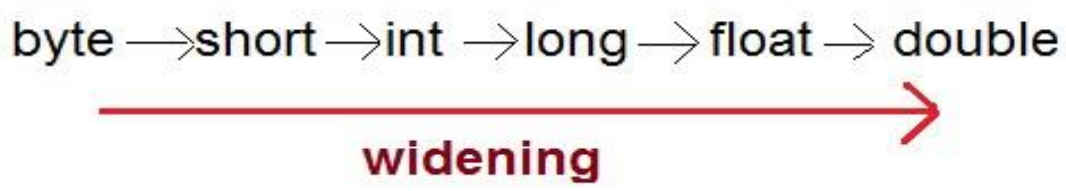
Casting is a process of changing one type value to another type. In Java, we can cast one type of value to another type. It is known as type casting.

Example :

```
int x = 10;
byte y = (byte)x;
```

In Java, type casting is classified into two types,

1. Widening Casting(Implicit)



2. Narrowing Casting(Explicitly done)

Widening or Automatic type converion

- Automatic Type casting take place when,
- the two types are compatible
- the target type is larger than the source type

Example:

```
public class Test
{    public static void main(String[] args)
   {     int i = 100;
     long l = i; //no explicit type casting required
     float f = l;  //no explicit type casting required
     System.out.println("Int value "+i);
```

```
    System.out.println("Long value "+l);
    System.out.println("Float value "+f);
  } }
```

Output:

Int value 100
Long value 100
Float value 100.0

Narrowing or Explicit type conversion

- When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting. If we don't perform casting then compiler reports compile time error.

Example :

```
public class Test
{    public static void main(String[] args)
  {      double d = 100.04;
    long l = (long)d;  //explicit type casting required
    int i = (int)l; //explicit type casting required
    System.out.println("Double value "+d);
    System.out.println("Long value "+l);
    System.out.println("Int value "+i);
  }
 }
```

Output:

Double value 100.04
Long value 100
Int value 100

Java Comments

The Java comments are the statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code.

Types of Java Comments

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1. Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

```
//This is single line comment
```

Example:

```
public class CommentExample1 {
```

```
public static void main(String[] args) {
   int i=10;//Here, i is a variable
   System.out.println(i);
}  }
```

Output:

10

2.Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
/*
This
is
multi line
comment
*/
```

Example:

```
public class CommentExample2 {
public static void main(String[] args) {
/* Let's declare and
 print variable in java. */
   int i=10;
   System.out.println(i);  }  }
```

Output:

10

3.Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax:

```
/**
This
is
documentation
comment
*/
```

Example:

```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
public class Calculator {
/** The add() method returns addition of given numbers.*/
public static int add(int a, int b){return a+b;}
/** The sub() method returns subtraction of given numbers.*/
```

```
public static int sub(int a, int b){return a-b;}
}
```

Compile it by javac tool:

```
javac Calculator.java
```

Create Documentation API by javadoc tool:

```
javadoc Calculator.java
```

Control Statements in Java

- Control Statements in Java is one of the fundamentals required for Java Programming. It allows the smooth flow of a program. Following pointers will be covered in this article:
- Decision Making Statements
    - Simple if statement
    - if-else statement
    - Nested if statement
    - Switch statement
- Looping statements
    - While
    - Do-while
    - For
    - For-Each
- Branching statements
    - Break
    - Continue

Control Statements can be divided into three categories, namely

1. Selection statements
2. Iteration statements
3. Jump statements

Decision making

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

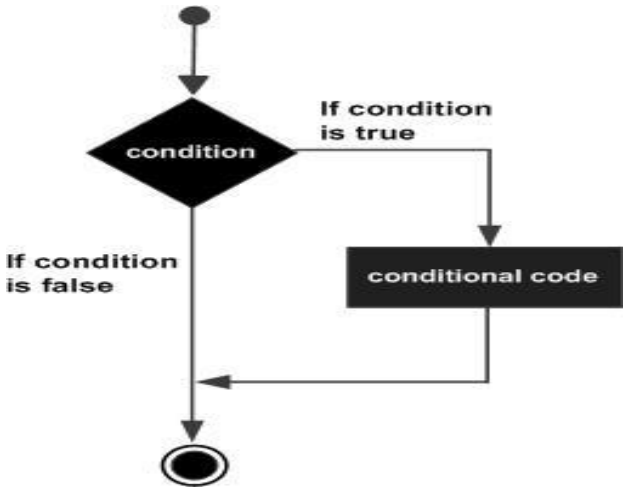
if statement

An if statement consists of a Boolean expression followed by one or more statements.

Syntax

```
if(Boolean_expression) {
  // Statements will execute if the Boolean expression is true
}
```

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    int x = 10;
    if( x < 20 ) {
      System.out.print("This is if statement");
    }  }}
```

Output

This is if statement.

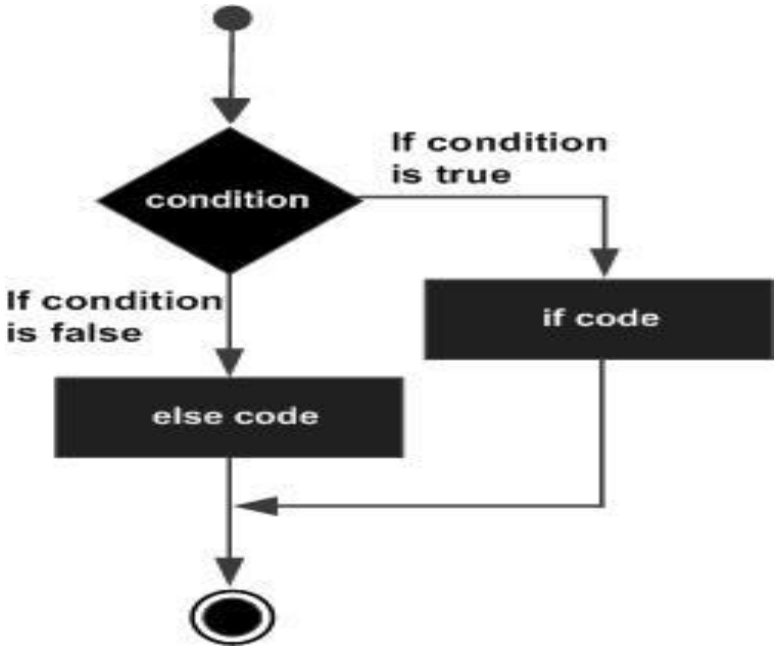
if-else statement

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax

```
if(Boolean_expression) {
  // Executes when the Boolean expression is true
}else {
  // Executes when the Boolean expression is false
}
```

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    int x = 30;
    if( x < 20 ) {
      System.out.print("This is if statement");
    }else {
      System.out.print("This is else statement");
    }  }}
```

Output

This is else statement

The if...else if...else Statement

- An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.
- When using if, else if, else statements there are a few points to keep in mind.
- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

```
if(Boolean_expression 1) {
  // Executes when the Boolean expression 1 is true
```

```
}else if(Boolean_expression 2) {
   // Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3) {
   // Executes when the Boolean expression 3 is true
}else {
   // Executes when the none of the above condition is true.
}
```

Example

```
public class Test {
  public static void main(String args[]) {
    int x = 30;
    if( x == 10 ) {
      System.out.print("Value of X is 10");
    }else if( x == 20 ) {
      System.out.print("Value of X is 20");
    }else if( x == 30 ) {
      System.out.print("Value of X is 30");
    }else {
      System.out.print("This is else statement");
    }   } }
```

Output

Value of X is 30

Nested if statement

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

Syntax

```
if(Boolean_expression 1) {
  // Executes when the Boolean expression 1 is true
  if(Boolean_expression 2) {
    // Executes when the Boolean expression 2 is true
  }}
```

Example

```
public class Test {
  public static void main(String args[]) {
    int x = 30;
    int y = 10;
```

```
    if( x == 30 ) {
      if( y == 10 ) {
        System.out.print("X = 30 and Y = 10");
      }     }   }}
```

Output

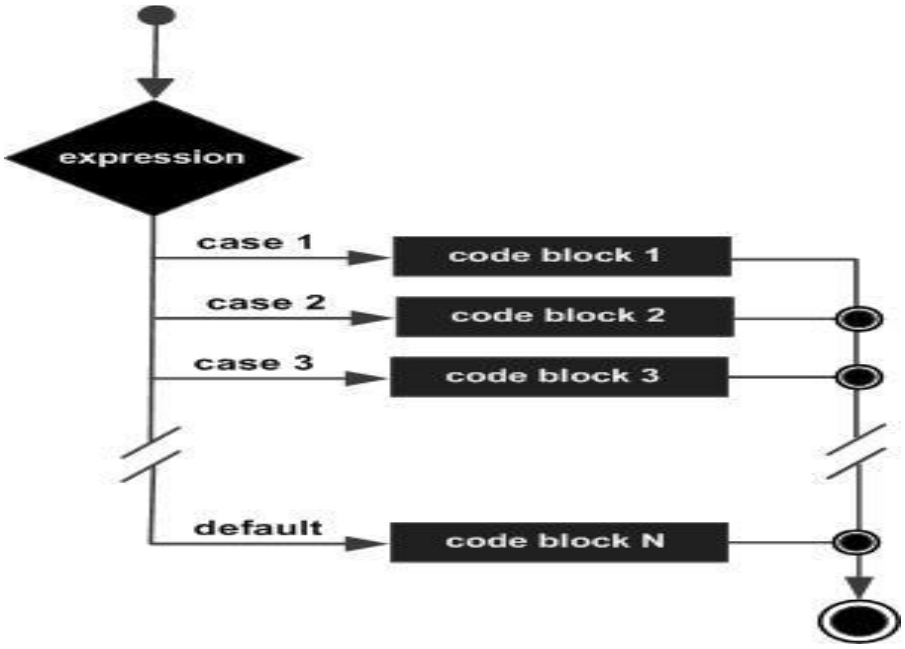
X = 30 and Y = 10

Switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

```
switch(expression) {
   case value :
      // Statements
      break; // optional
      case value :
      // Statements
      break; // optional
      // You can have any number of case statements.
   default : // Optional
      // Statements
}
```



The following rules apply to a **switch** statement −

- The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example

```
public class Test {
  public static void main(String args[]) {
    // char grade = args[0].charAt(0);
    char grade = 'C';
    switch(grade) {
      case 'A' :
        System.out.println("Excellent!");
        break;
      case 'B' :
      case 'C' :
        System.out.println("Well done");
        break;
      case 'D' :
        System.out.println("You passed");
      case 'F' :
        System.out.println("Better try again");
        break;
      default :
        System.out.println("Invalid grade");      }
    System.out.println("Your grade is " + grade);
  }}
```

Output

Well done

Your grade is C

Loop Control

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages

while Loop

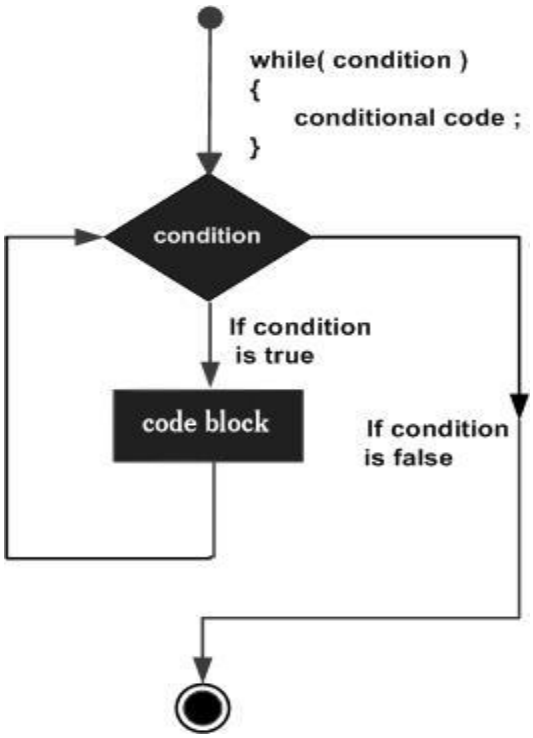
A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

```
while(Boolean_expression)
 {
   // Statements
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

Flow Diagram



Example

```
public class Test {
   public static void main(String args[]) {
      int x = 10;
      while( x < 20 ) {
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");       }    }}
```

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

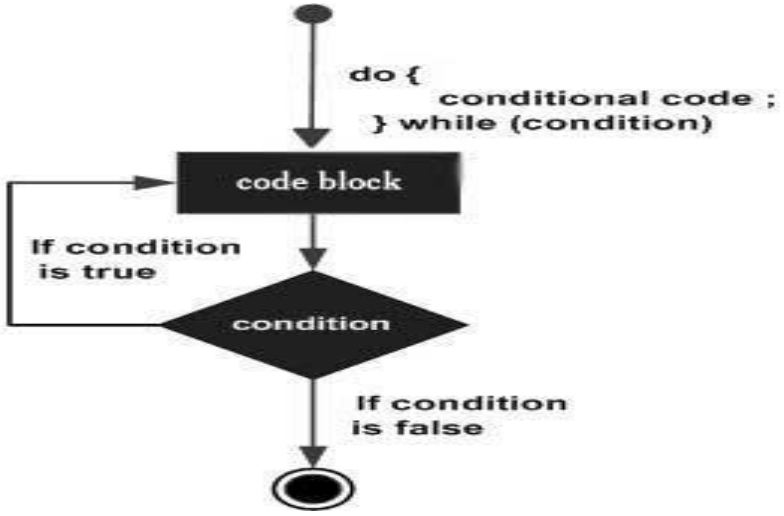
do while loop

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

```
Do
 {
   // Statements
}while(Boolean_expression);
```

Flow Diagram



Example

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    do {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }while( x < 20 );
  }}
```

This will produce the following result −

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

for loop

- A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.
- A **for** loop is useful when you know how many times a task is to be repeated.

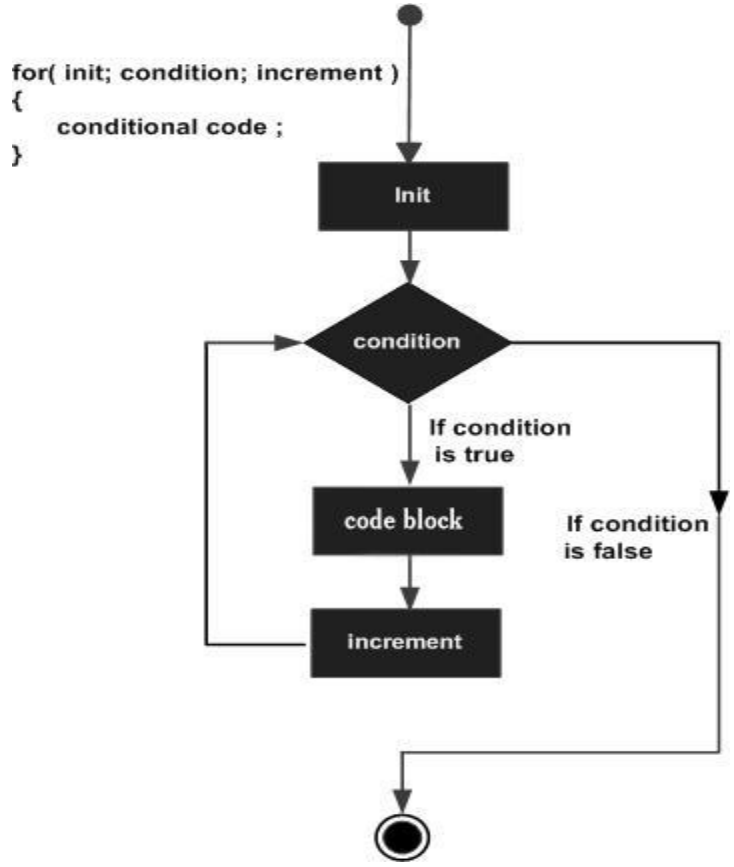
Syntax

```
for(initialization; Boolean_expression; update)
 {
   // Statements
}
```

Here is the flow of control in a **for** loop −

- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
- Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
- After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    for(int x = 10; x < 20; x = x + 1) {
      System.out.print("value of x : " + x );
      System.out.print("\n");
    }  }}
```

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

Break statement

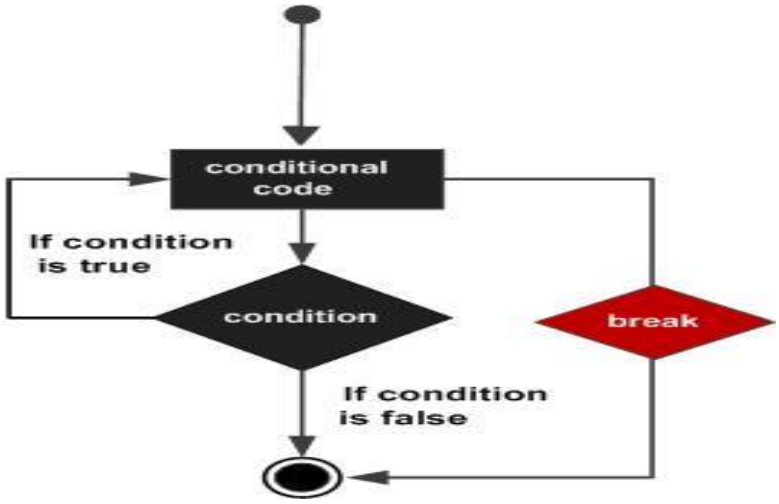
The **break** statement in Java programming language has the following two usages

- When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

Syntax

```
break;
```

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};
    for(int x : numbers ) {
      if( x == 30 ) {
        break;
      }
      System.out.print( x );
      System.out.print("\n");
    }  }}
```

Output

```
10
20
```

Continue statement

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

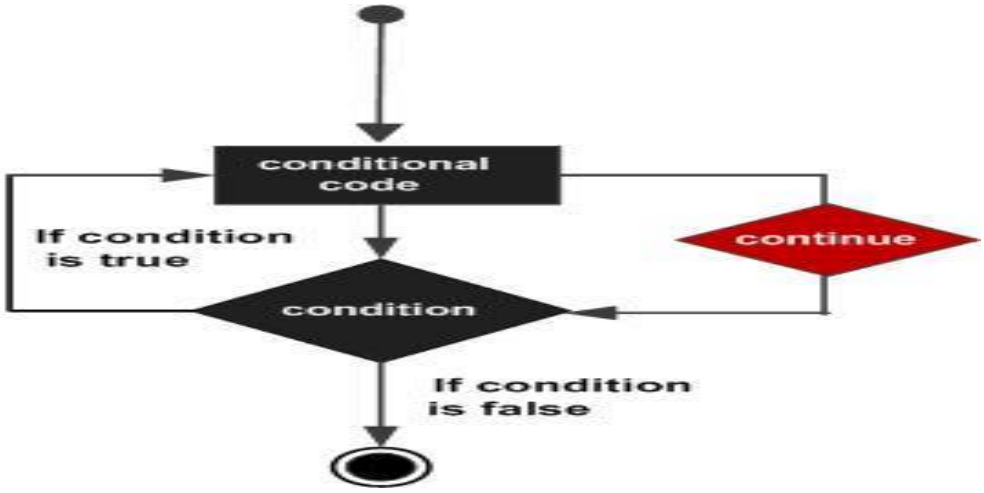
- In a for loop, the continue keyword causes control to immediately jump to the update statement.

- In a while loop or do/while loop, control immediately jumps to the Boolean expression.

Syntax

continue;

Flow Diagram



Example

```
public class Test {
   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};
      for(int x : numbers ) {
         if( x == 30 ) {
            continue;
         }
         System.out.print( x );
         System.out.print("\n");
      }   }}
```

Output

10
20
40
50

Class and Object

Classes and objects are the fundamental components of OOP's. Often there is a confusion between classes and objects.

Class

CLASS are a blueprint or a set of instructions to build a specific type of object. It is a basic concept of Object-Oriented Programming which revolve around the real-life entities. Class in Java determines how an object will behave and what the object will contain.

Syntax

```
class <class_name>
{
   field;
   method;
  }
```

Object

OBJECT is an instance of a class. An object is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful. For example color name, table, bag, barking. When you send a message to an object, you are asking the object to invoke or execute one of its methods as defined in the class.From a programming point of view, an object can include a data structure, a variable, or a function. It has a memory location allocated. The object is designed as class hierarchies.

Syntax

```
ClassName ReferenceVariable = new ClassName();
```

Difference Between Object & Class

- A **class** is a **blueprint or prototype** that defines the variables and the methods (functions) common to all objects of a certain kind.
- An **object** is a specimen of a class. Software objects are often used to model real-world objects you find in everyday life.

Example Code: Class and Object

```
/ Class Declaration
class Dog {
   // Instance Variables
   String breed;
   String size;
   int age;
   String color;
   // method 1
   public String getInfo() {
      return ("Breed is: "+breed+" Size is:"+size+" Age is:"+age+" color is: "+color);
   }  }
public class Execute{
   public static void main(String[] args) {
      Dog maltese = new Dog();
```

```
    maltese.breed="Maltese";
    maltese.size="Small";
    maltese.age=2;
    maltese.color="white";
    System.out.println(maltese.getInfo());
  } }
```

Output:

Breed is: Maltese Size is:Small Age is:2 color is: white

Access Modifiers

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifier	within class	within package	**outside package by subclass only**	**outside package**
Private	Y	N	**N**	N
Default	Y	Y	**N**	N
Protected	Y	Y	**Y**	N
Public	Y	Y	**Y**	Y

Private

- The private access modifier is accessible only within the class.

Example

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}
 public class Simple{
public static void main(String args[]){
  A obj=new A();
  System.out.println(obj.data);//Compile Time Error
  obj.msg();//Compile Time Error
  }  }
```

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }  }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
  class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }  }
```

Output: Hello

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java
  package pack;
public class A{
public void msg(){System.out.println("Hello");}  }
//save by B.java
  package mypack;
import pack.*;
  class B{
  public static void main(String args[]){
   A obj = new A();
```

```
  obj.msg();
 } }
```

Output:Hello

JAVA Parameters and Arguments

- Information can be passed to methods as parameter. Parameters act as variables inside the method.
- Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.
- The following example has a method that takes a String called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

MyClass.java

```
public class MyClass {
 static void myMethod(String fname) {
  System.out.println(fname + " Refsnes");
 }
 public static void main(String[] args) {
  myMethod("Liam");
  myMethod("Jenny");
  myMethod("Anja");
 }
```

OUTPUT

```
Liam Refsnes
Jenny Refsnes
Anja Refsnes
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: fname is a **parameter**, while Liam, Jenny and Anja are **arguments**.

Multiple Parameters

MyClass.java

```
public class MyClass {
 static void myMethod(String fname, int age) {
  System.out.println(fname + " is " + age);
 }
 public static void main(String[] args) {
  myMethod("Liam", 5);
```

```
  myMethod("Jenny", 8);
  myMethod("Anja", 31);
 }
```

OUTPUT

liam is 5
Jenny is 8
Anja is 31

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

MyClass.java

```
public class MyClass {
 static int myMethod(int x) {
  return 5 + x;
 }
 public static void main(String[] args) {
  System.out.println(myMethod(3));
 }}
```

OUTPUT

8

Constructors in Java

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
- There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Rules for creating Java constructor

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

1. **Default constructor (no-arg constructor)**
2. **Parameterized constructor**

Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
<class_name>(){}
```

Example

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
}  }
```

Output:

```
Bike is created
```

Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Example

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
   int id;
   String name;
   //creating a parameterized constructor
   Student4(int i,String n){
   id = i;
   name = n;
   }
   //method to display the values
   void display(){System.out.println(id+" "+name);}
     public static void main(String args[]){
   //creating objects and passing values
   Student4 s1 = new Student4(111,"Karan");
   Student4 s2 = new Student4(222,"Aryan");
   //calling method to display the values of object
```

```
   s1.display();
   s2.display();
  }  }
```

Output:

111 Karan
222 Aryan

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
   int id;
   String name;
   int age;
   //creating two arg constructor
   Student5(int i,String n){
   id = i;
   name = n;
   }
   //creating three arg constructor
   Student5(int i,String n,int a){
   id = i;
   name = n;
   age=a;
   }
   void display(){System.out.println(id+" "+name+" "+age);}
     public static void main(String args[]){
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
  }  }
```

Output:

111 Karan 0
222 Aryan 25

Java Packages & API

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

- The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.
- The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles.
- The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
- To use a class or a package from the library, you need to use the `import` keyword:

Syntax

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the `Scanner` class, **which is used to get user input**, write the following code:

Example

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

- In the example above, `java.util` is a package, while `Scanner` is a class of the `java.util` package.
- To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read a complete line:

Example

Using the `Scanner` class to get user input:

```
import java.util.Scanner;
class MyClass
 {   public static void main(String[] args)
{
   Scanner myObj = new Scanner(System.in);
   System.out.println("Enter username");
   String userName = myObj.nextLine();
   System.out.println("Username is: " + userName);
```

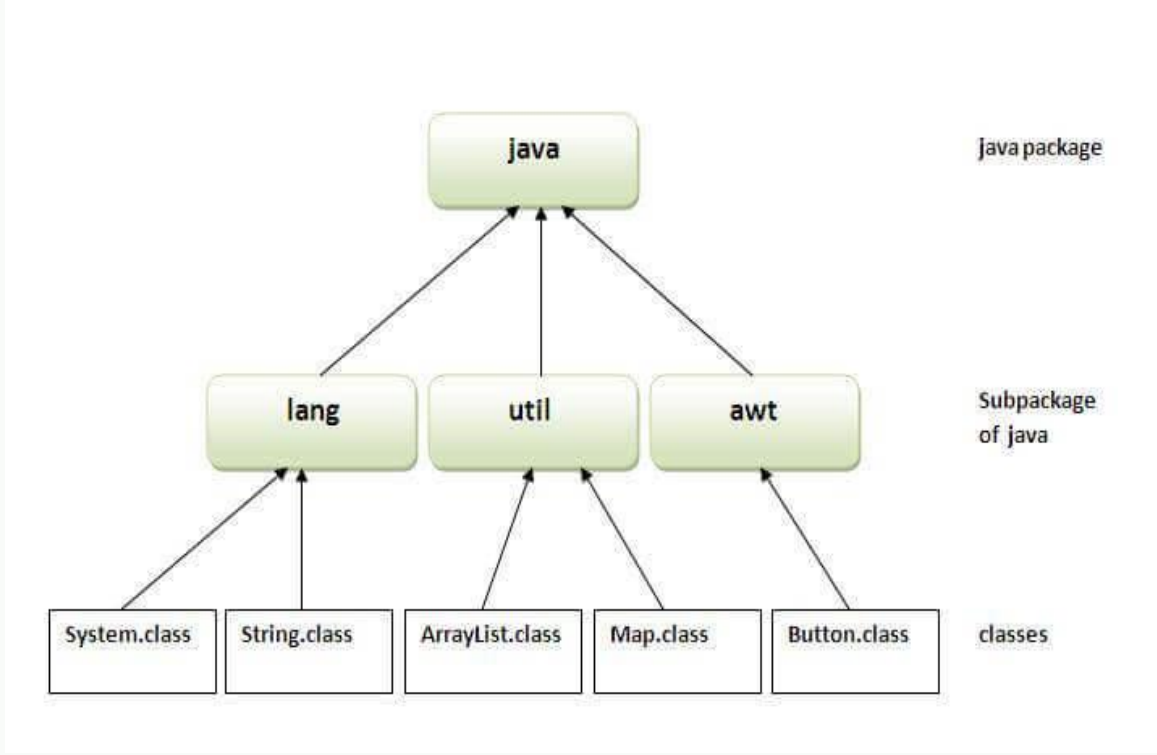
```
    }
}
```

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



How to compile java package

If you are not using any IDE, you need to follow the syntax given below:

javac -d directory javafilename

For example

javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output: Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?
There are three ways to access the package from outside the package.
import package.*;
import package.classname;
fully qualified name.

static class in Java

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class.

Syntax

```
class MyOuter {
  static class Nested_Demo {
  }}
```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

Example

```
class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }
}
```

Output

data is 30

Method Overloading

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

For example: This is a valid case of overloading
add(int, int)
add(int, int, int)

2. Data type of parameters.

For example:

add(int, int)

add(int, float)

3. Sequence of Data type of parameters.

For example:

add(int, float)

add(float, int)

Example 1: Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class DisplayOverloading
{    public void disp(char c)
    {        System.out.println(c);
    }
    public void disp(char c, int num)
    {        System.out.println(c + " "+num);
    }
}class Sample
{   public static void main(String args[])
   {
      DisplayOverloading obj = new DisplayOverloading();
      obj.disp('a');
      obj.disp('a',10);
   }}
```

Output:

a

a 10

Passing and Returning Objects

Although Java is strictly pass by value, the precise effect differs between whether a primitive type or a reference type is passed.

When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference. Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.

- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect in the object used as an argument.

```
// Java program to demonstrate objects
// passing to methods.
class ObjectPassDemo
{     int a, b;
      ObjectPassDemo(int i, int j)
    {         a = i;
        b = j;     }
      // return true if o is equal to the invoking
    // object notice an object is passed as an
    // argument to method
    boolean equalTo(ObjectPassDemo o)
    {         return (o.a == a && o.b == b);     } }
  // Driver class
public class Test
{     public static void main(String args[])
    {
        ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);
          System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    } }
```

Output:

```
ob1 == ob2: true
ob1 == ob3: false
```

this keyword

In Java, this is a keyword which is **used to refer current object** of a class. we can it to refer any member of the class. It means we can access any instance variable and method by using **this** keyword.The main purpose of using **this** keyword is to solve the confusion when we have same variable name for instance and local variables.We can use this keyword for the following purpose.

- **this** keyword is used to refer to current object.
- **this** is always a reference to the object on which method was invoked.
- **this** can be used to invoke current class constructor.
- **this** can be passed as an argument to another method.

- Lets first understand the most general use of this keyword. As we said, it can be used to differentiate local and instance variables in the class.

Example:

```
class Demo
{
 Double width, height, depth;
 Demo (double w, double h, double d)
 {
  this.width = w;
  this.height = h;
  this.depth = d;
 }
 public static void main(String[] args) {
          Demo d = new Demo(10,20,30);
          System.out.println("width = "+d.width);
          System.out.println("height = "+d.height);
          System.out.println("depth = "+d.depth); }}
```

OUTPUT

```
width = 10.0
height = 20.0
depth = 30.0
```

Enumeration

- Enumerations serve the purpose of representing a group of named constants in a programming language. For example the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).
  Enums are used when we know all possible values at **compile time**, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay **fixed** for all time.
- In Java (from 1.5), enums are represented using **enum** data type. Java enums are more powerful than C/C++ enums . In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our own data types(Enumerated Data Types).

Declaration of enum in java :

Enum declaration can be done outside a Class or inside a Class but not inside a Method.

```
// A simple enum example where enum is declared
// outside any class (Note enum keyword instead of
// class keyword)
```

```
enum Color
{     RED, GREEN, BLUE; }
  public class Test
{     // Driver method
    public static void main(String[] args)
    {         Color c1 = Color.RED;
       System.out.println(c1);
    } }
```

Output :

RED

Garbage Collection

- 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

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.

How can an object be unreferenced?

There are many ways:

1. **By nulling the reference**
2. **By assigning a reference to another**
3. **By anonymous object etc.**

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

3) By anonymous object:

new Employee();

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.

xample

```
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

Text Book

1. S.Sagayaraj, R.Denis, P.Karthik & D.Gajalakshmi, “Java Programming“, Universities Press, 2017

References

1. Patrick Naughton and Herbert Schildt. “The Complete Reference JAVA 2”. 3rd Edition. Tata McGraw-Hill Edition, 1999.
2. Muthu C. “Programming with JAVA”. 2nd Edition. Vijay Nicole Imprints, 2011.
3. Ken Arnold Gosling and Davis Holmen. “The Java Programming Language”. 3rd Edition. Addition Wesley Publication.