BEIT JAVA

Programming in Java (Pokhara University)

Chapter 1

Java as a programming tool

Java is a high-level programming language that is widely used for developing complex software applications. It was first released in 1995 by Sun Microsystems and has since become one of the most popular programming languages in the world.

Java is known for its "write once, run anywhere" capability, which means that Java code can be run on any platform without the need for any modifications. This is because Java code is compiled into an intermediate bytecode that can be interpreted by any Java Virtual Machine (JVM) on any platform.

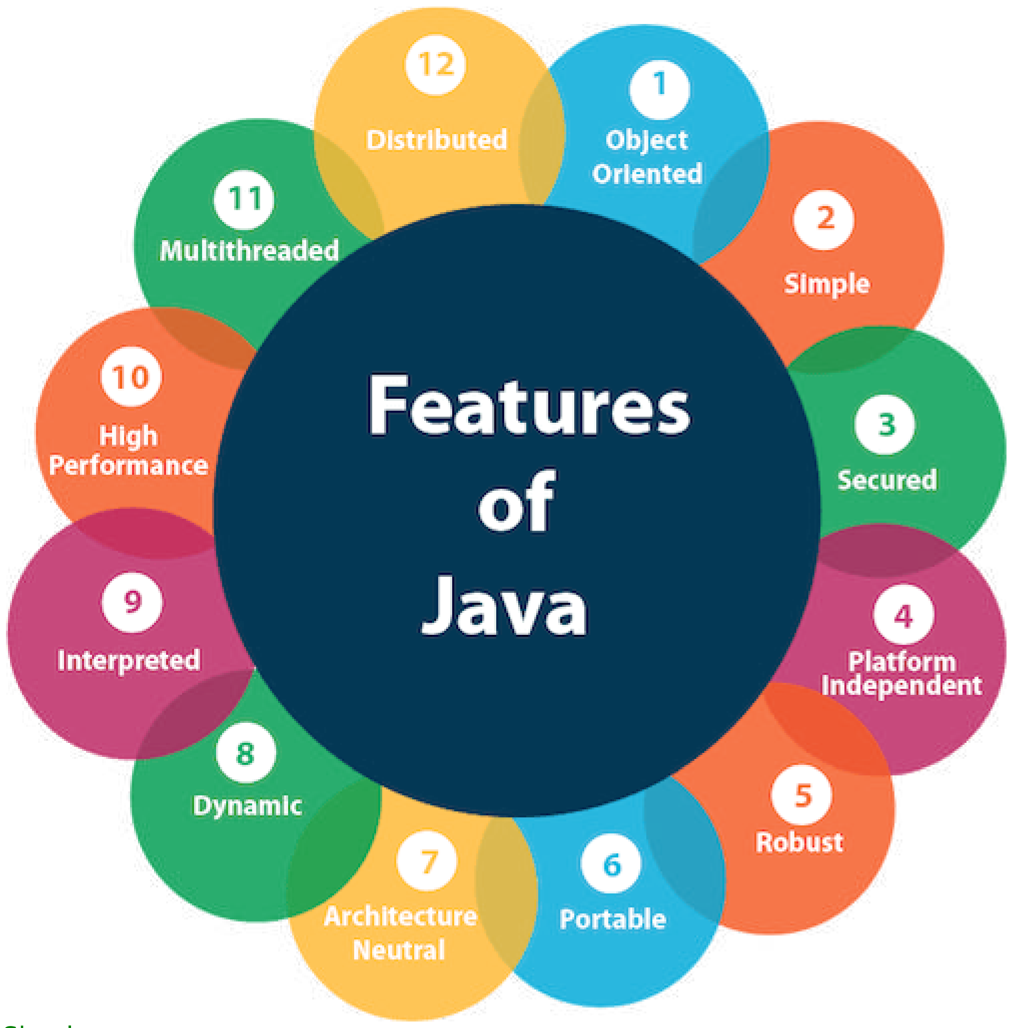
Java is an object-oriented language, which means that it allows developers to create reusable code in the form of objects and classes. Java also supports multithreading, which means that multiple threads of execution can run concurrently within a single program.

Java has a rich set of built-in libraries that provide functionality for tasks such as input/output, networking, database connectivity, and user interface development. Additionally, Java has a large and active community of developers who contribute to open source libraries and frameworks, making it easy for developers to find and use third-party libraries and tools.

Overall, Java is a powerful and versatile programming tool that is widely used in industries such as finance, healthcare, and e-commerce, among others. It has a strong focus on object-oriented programming, portability, and ease of use, which makes it an ideal choice for developing complex software applications.

Benefits of Java

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



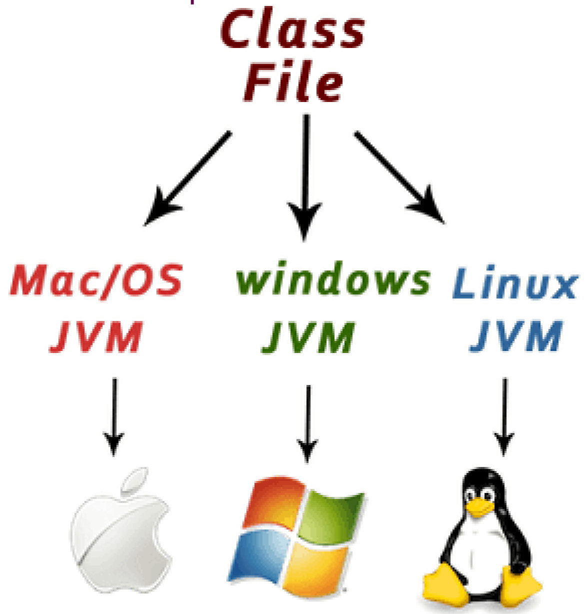
Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, 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 incorporate both data and behavior.
Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.
Basic concepts of OOPs are:
Object
Class
Inheritance
Polymorphism
Abstraction
Encapsulation

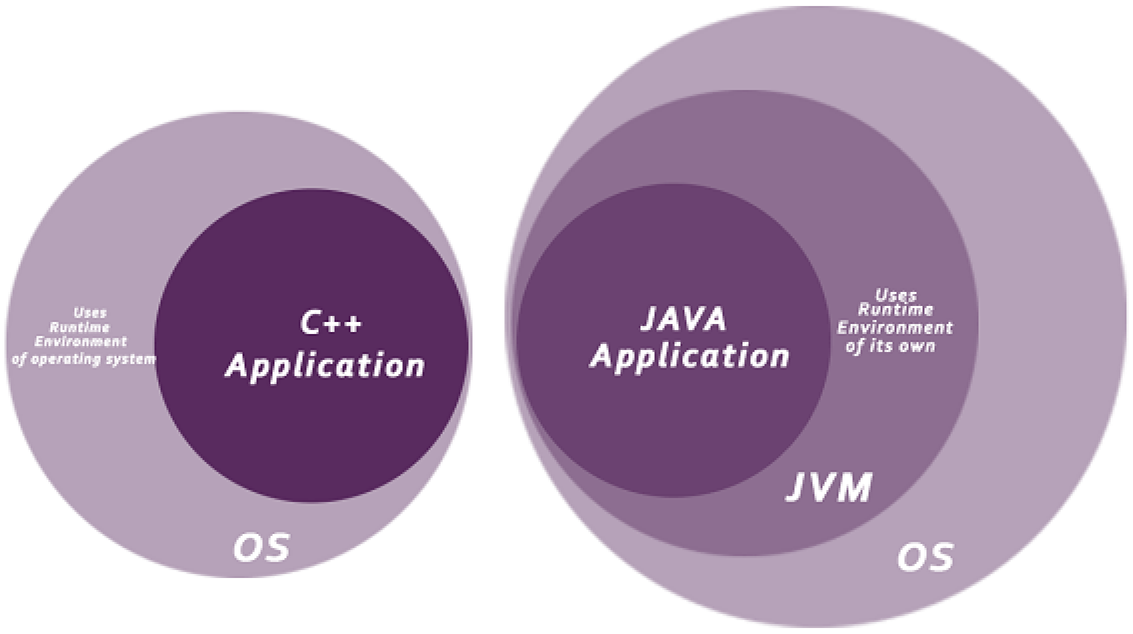
Platform Independent



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 top of other hardware-based platforms. It has two components:
Runtime Environment
API(Application Programming Interface)
Java code can be executed 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


Uses Runtime Environment of operating system
C++ Application
OS
JAVA Application
Uses Runtime Environment of its own
JVM
OS

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 rights 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

The English mining of Robust is strong. Java is robust because:
It uses strong memory management.
There is a lack of pointers that avoids security problems.
Java provides automatic garbage collection 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

Architecture-neutral means that a Java program can run on any machine or operating system regardless of its underlying architecture or hardware configuration. This is possible because Java has been designed to eliminate implementation-dependent features such as the size of primitive data types, which vary depending on the architecture in the case of C programming. In Java, the size of primitive data types is fixed, so the same code will work on different machines and operating systems.
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 the 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).

Hello world

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

Data Types in Java

```
public class AllDataTypes {

  public static void main(String[] args) {

    boolean boolVar = true;

    byte byteVar = 127;
    short shortVar = 32767;

    int intVar = 2147483647;

    long longVar = 9223372036854775807L;

    float floatVar = 3.4028235E38f;

    double doubleVar = 1.7976931348623157E308;

    char charVar = 'A';

    String stringVar = "Hello, world!";


    System.out.println("boolean: " + boolVar);

    System.out.println("byte: " + byteVar);

    System.out.println("short: " + shortVar);

    System.out.println("int: " + intVar);

    System.out.println("long: " + longVar);

    System.out.println("float: " + floatVar);

    System.out.println("double: " + doubleVar);

    System.out.println("char: " + charVar);

    System.out.println("String: " + stringVar);

  }

}
```

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.

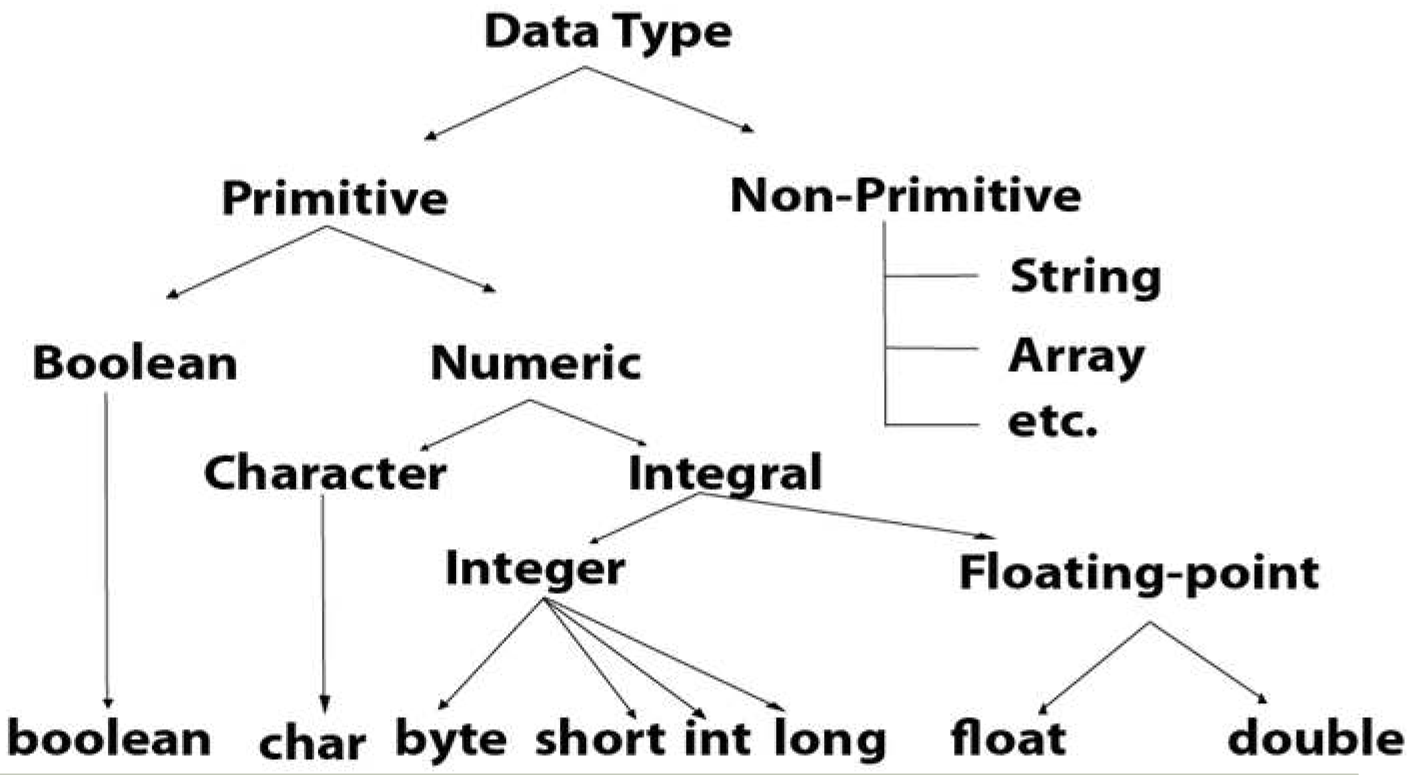
Java Primitive Data Types

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:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- 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

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:

1. Boolean one = **false**

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:

1. **byte** a = 10, **byte** b = -20

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:

1. `short s = 10000, short r = -5000`

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:

1. `int a = 100000, int b = -200000`

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:

1. `long a = 100000L, long b = -200000L`

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:

1. `float f1 = 234.5f`

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:

```
1.          double d1 = 12.3
```

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:

```
1.          char letterA = 'A'
```

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

Variable

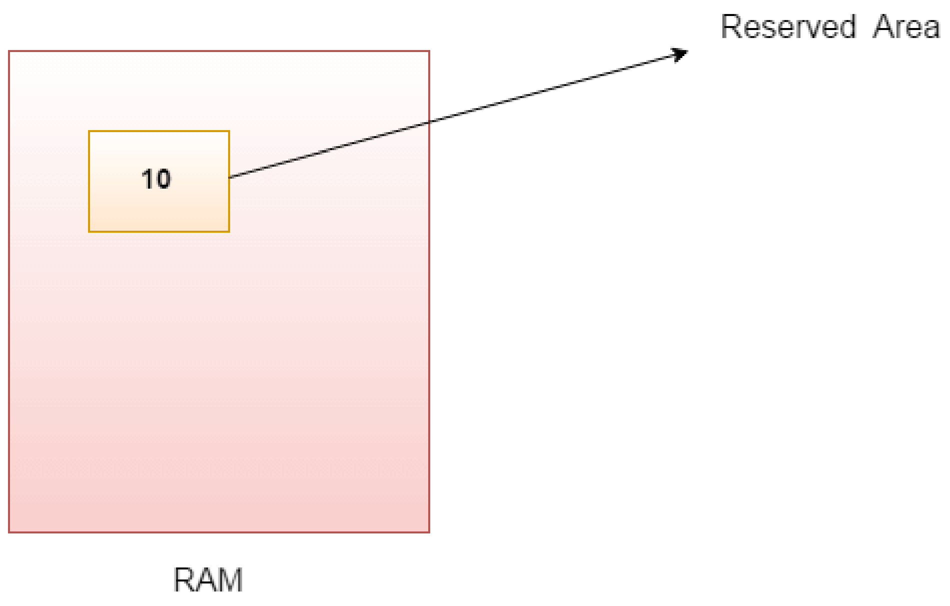
A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in Java: primitive and non-primitive.

Variable

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

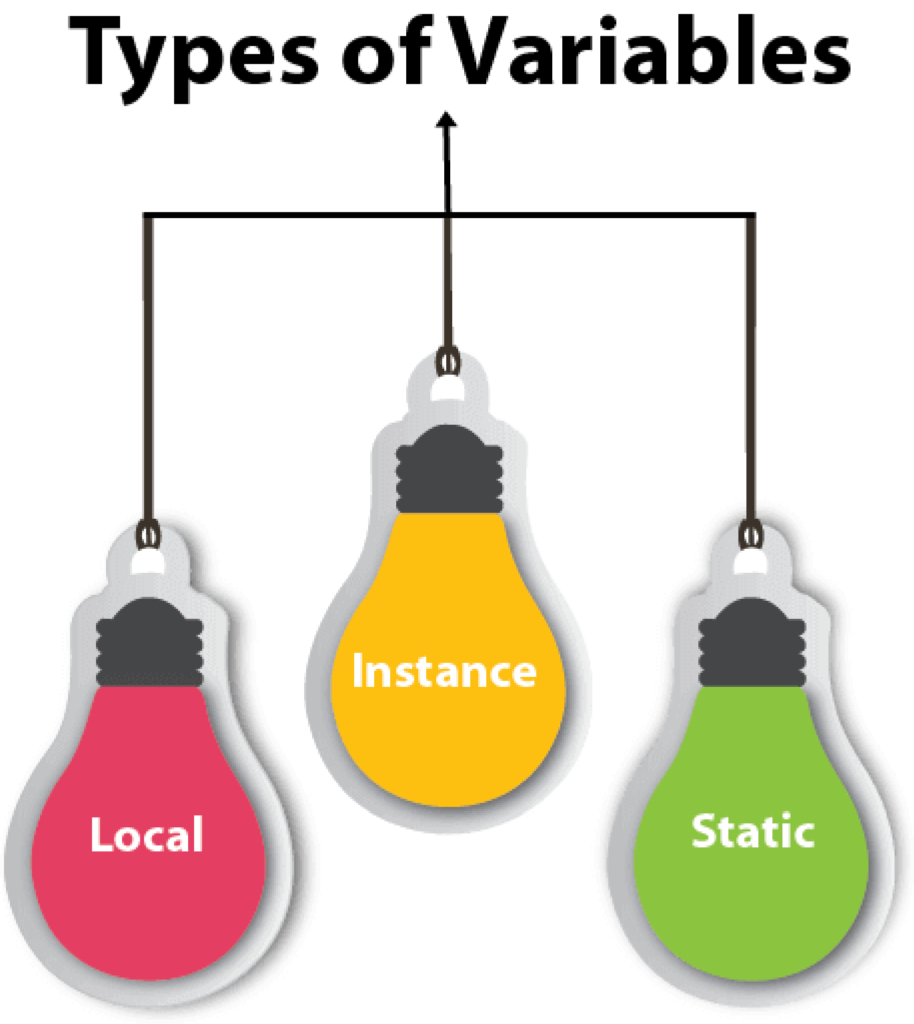


```
1.          int data=50;//Here data is variable
```

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- 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 an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

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

Example to understand the types of variables in java

```
public class A
{
   static int m=100;//static variable
   void method()
   {
      int n=90;//local variable
   }
   public static void main(String args[])
   {
      int data=50;//instance variable
   }
}//end of class
```

Java Variable Example: Add Two Numbers

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

Output:

```
20
```

Java Variable Example: Widening

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

Output:

```
10
10.0
```

Java Variable Example: Narrowing (Typecasting)

```
public class Simple{
public static void main(String[] args){
float f=10.5f;
//int a=f;//Compile time error
int a=(int)f;
System.out.println(f);
System.out.println(a);
}}
```

Output:

```
10.5
10
```

Java Variable Example: Overflow

```
class Simple{
public static void main(String[] args){
//Overflow
int a=130;
byte b=(byte)a;
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
130
-126
```

Java Variable Example: Adding Lower Type

```
class Simple{
public static void main(String[] args){
byte a=10;
byte b=10;
//byte c=a+b;//Compile Time Error: because a+b=20 will be int
byte c=(byte)(a+b);
System.out.println(c);
}}
```

Assignment and initialization
In Java, assignment is the process of setting a value to a variable, while initialization is the process of declaring and assigning a value to a variable at the same time.
Here is an example of assignment:
int x; x = 5; // value 5 is assigned to variable x
And here is an example of initialization:
int y = 10; // variable y is declared and assigned the value 10 at the same time
In the initialization example, the variable "y" is declared and assigned the value of 10 in a single statement. This is useful when you want to declare and assign a value to a variable at the same time.

Operator
In Java, operators are special symbols or characters that are used to perform operations on variables and values. There are different types of operators in Java, such as arithmetic operators, comparison operators, logical operators, bitwise operators, and assignment operators.

1. Arithmetic operators: Arithmetic operators are used to perform mathematical operations, such as addition, subtraction, multiplication, division, and modulus. For example, +, -, *, /, and %.
2. Comparison operators: Comparison operators are used to compare two values and return a Boolean value (true or false). For example, ==, !=, >, >=, <, and <=.
3. Logical operators: Logical operators are used to combine two or more Boolean expressions and return a Boolean value. For example, && (AND), || (OR), and ! (NOT).
4. Bitwise operators: Bitwise operators are used to perform bitwise operations on integer values. For example, &, |, ^, ~, <<, and >>.
5. Assignment operators: Assignment operators are used to assign a value to a variable. For example, =, +=, -=, *=, /=, and %=.

Example: int a = 5; int b = 3; int c = a + b; //
addition operator boolean isTrue = (a > b) && (a != b);
// logical AND operator int d = a << 2;
// left shift operator a += b;
// shorthand assignment operator for addition

String
In Java, String is a class that represents a sequence of characters. It is used to store and manipulate textual data. String objects are immutable, which means that their values cannot be changed once they are created.
You can create a String object by enclosing a sequence of characters in double quotes, like this:
arduinoCopy code
String myString = "Hello, world!";
You can also create a String object using the **new** keyword and passing an array of characters to the constructor, like this:

arduinoCopy code

```
char[] myCharArray = { 'H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!' };
String myString = new String(myCharArray);
```

The String class provides many useful methods for manipulating strings, such as **length()**, **charAt()**, **substring()**, **toUpperCase()**, **toLowerCase()**, **trim()**, **startsWith()**, **endsWith()**, **contains()**, and many more.

Control flow

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
   - if statements
   - switch statement
2. Loop statements
   - do while loop
   - while loop
   - for loop
   - for-each loop
3. Jump statements
   - break statement
   - continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder

4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {
statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the **if** statement in the java code.

Student.java

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Consider the following example.

Student.java

```java
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
}
}
}
```

Output:

```
Delhi
```

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```java
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Consider the following example.

Student.java

```java
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}
}
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- *The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java*
- *Cases cannot be duplicate*
- *Default statement is executed when any of the case doesn't match the value of expression. It is optional.*
- *Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.*
- *While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a* constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
    case value1:
     statement1;
     break;
    .
    .
```

```
                .
                case valueN:
                 statementN;
                 break;
                default:
                 default statement;
            }
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output:

```
2
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop

2. while loop
3. do-while loop

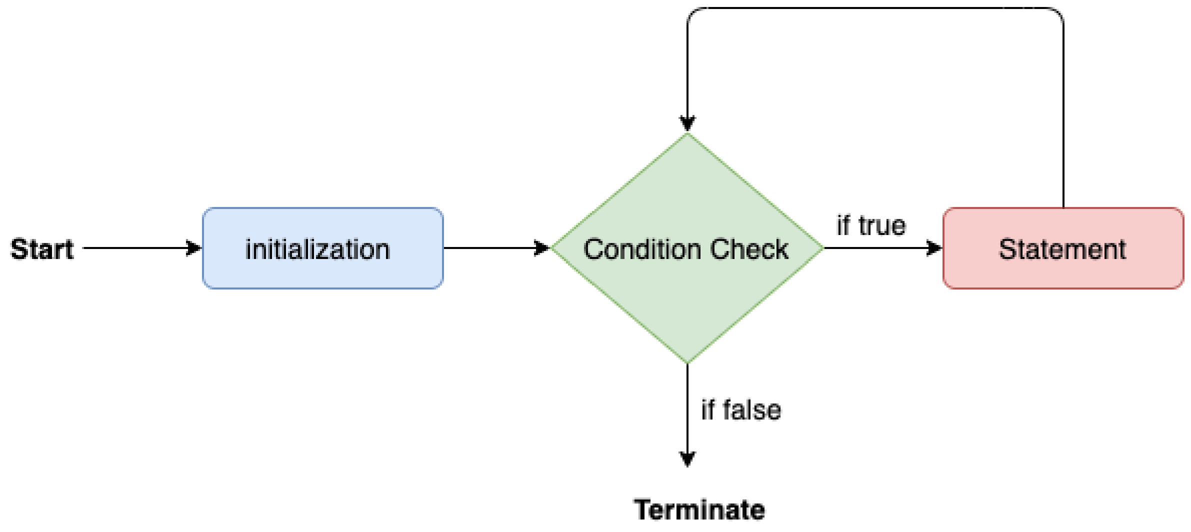
Let's understand the loop statements one by one.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);
}
```

```java
}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```java
for(data_type var : array_name/collection_name){
//statements
}
```

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```java
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

Java while loop

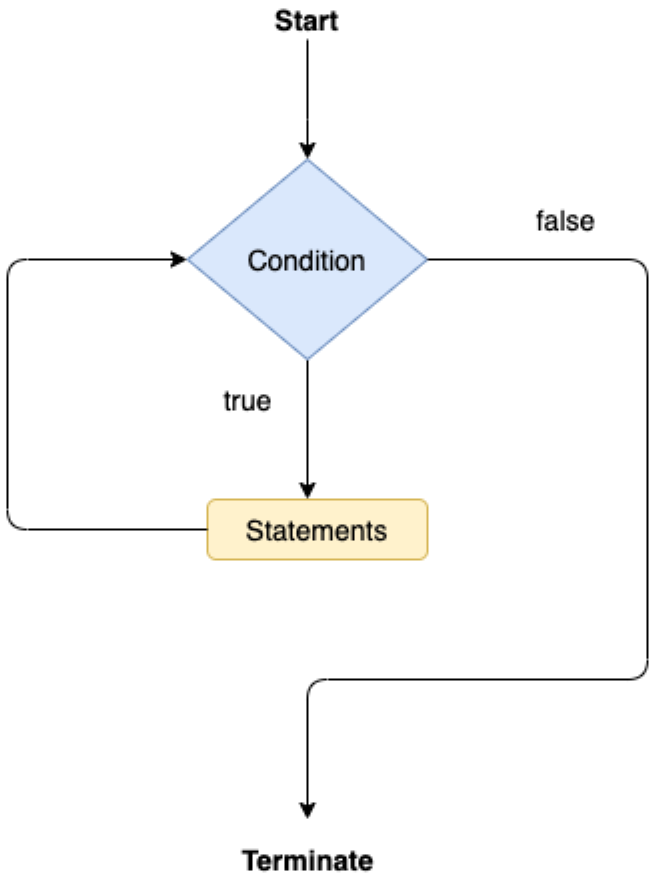
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```
while(condition){
//looping statements
}
```

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

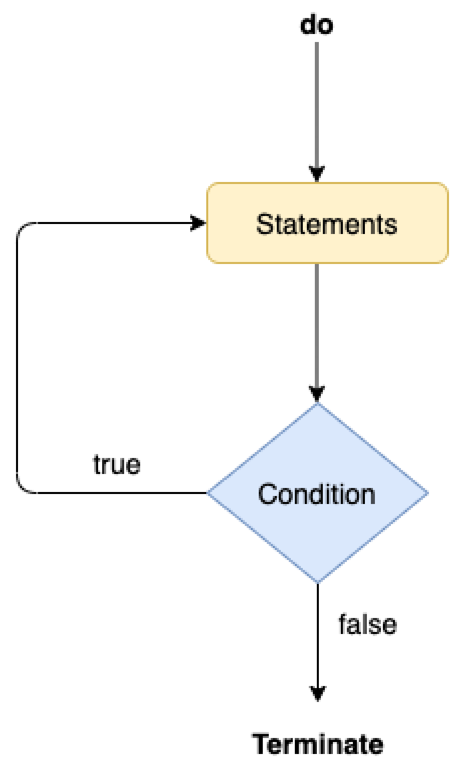
Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

```
do
{
//statements
} while (condition);
```

The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}
}
```

Output:

```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

BreakExample.java

```
public class BreakExample {

public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output:

```
0
1
2
3
4
5
6
```

break statement example with labeled for loop

Calculation.java

```
public class Calculation {
```

```java

public static void main(String[] args) {
// TODO Auto-generated method stub
a:
for(int i = 0; i<= 10; i++) {
b:
for(int j = 0; j<=15;j++) {
c:
for (int k = 0; k<=20; k++) {
System.out.println(k);
if(k==5) {
break a;
}
}
}

}
}


}
```

Output:

```
0
1
2
3
4
5
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```java
public class ContinueExample {

public static void main(String[] args) {
// TODO Auto-generated method stub

for(int i = 0; i<= 2; i++) {

for (int j = i; j<=5; j++) {
```

```
            if(j == 4) {
            continue;
            }
            System.out.println(j);
            }
            }
            }

            }
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

Java Modifiers Like other languages, it is possible to modify classes, methods, etc., by using modifiers.
There are two categories of modifiers:
Access Modifiers: default, public , protected, private
Non-access Modifiers: final, abstract, strictfp

1. **Access Specifier default**:

In Java, the **default** keyword is used to define the default access level of a class, method or variable. If no access modifier is specified, Java automatically sets it to **default**. This means that the class, method or variable can only be accessed within the same package.

```
// Default access modifier
class MyClass {
  int x = 5; // Default access modifier
}
```

In the above example, **MyClass** has a default access level, which means it can only be accessed within the same package.

In Java, **private** and **default** are both access modifiers that control the visibility of a class member (field or method) from other classes.
The main difference between **private** and **default** is the scope of visibility:

- **private**: A private member is only visible within the same class in which it is declared. It cannot be accessed from outside the class, not even from its subclasses. For example:

```
public class MyClass {
   private int myPrivateField;

   public void myMethod() {
      // myPrivateField can be accessed here
   }
}

public class MyOtherClass {
   public void myOtherMethod() {
      MyClass myObject = new MyClass();
      // myObject.myPrivateField is not visible here
   }
}
```

default (also called package-private): A default member is visible within the same package in which it is declared. It cannot be accessed from outside the package, but can be accessed from any class within the same package, including its subclasses. To declare a default member, simply omit any access modifier before its declaration. For example:

```
package mypackage;

public class MyClass {
   int myDefaultField;

   void myMethod() {
      // myDefaultField can be accessed here
   }
}

package mypackage;

public class MyOtherClass {
   void myOtherMethod() {
      MyClass myObject = new MyClass();
      // myObject.myDefaultField can be accessed here
   }
}
```

In summary, the main difference between **private** and **default** is that **private** members are visible only within the same class, while **default** members are visible within the same package.

2. **final**:

In Java, the **final** keyword is used to make a variable or method immutable, which means they cannot be modified once they are initialized. When applied to a class, it makes the class non-extendable.

```
// Final variable
final int x = 10;

// Final method
public final void myMethod() {
  // Code goes here
}

// Final class
final class MyClass {
  // Code goes here
}
```
In the above example, **x** is a final variable, which cannot be changed once it is initialized. **myMethod()** is a final method, which cannot be overridden in a subclass. **MyClass** is a final class, which cannot be extended.

3. **abstract**:

In Java, the **abstract** keyword is used to create abstract classes and methods. Abstract classes cannot be instantiated, and can only be subclassed. Abstract methods do not have a body, and must be implemented in a subclass.
Abstract: The abstract keyword is used to declare a class or method that has no implementation. An abstract class cannot be instantiated, and an abstract method must be implemented in a subclass.

Example:
```
// Abstract class
abstract class Shape {
  // Abstract method
  public abstract void draw();
}

// Subclass of Shape
class Circle extends Shape {
  // Implementing the draw() method
  public void draw() {
    // Code goes here
  }
}
```

In the above example, **Shape** is an abstract class, which cannot be instantiated. It has an abstract method **draw()**, which must be implemented in a subclass. **Circle** is a subclass of **Shape**, which implements the **draw()** method.

3. Static: The static keyword is used to create class-level variables and methods that can be accessed without creating an instance of the class

```
public class demo {
   int sum(int a,int b){
      int c = a+b;
      return c;
```

```
    }

    static int diff(int c,int d){
        int e = c-d;
        return e;
    }

    public static void main(String[] args) {
        demo demo = new demo();
        int total = demo.sum(10,20);

        int diff = diff(10,30);

        System.out.println(total);
        System.out.println(demo.sum(10,20));

        System.out.println(diff);
        System.out.println(diff(40,30));
    }
}
```

User Defined Function
In Java, a user-defined function is called a method. A method is a set of code statements that can be called to perform a specific task.
Here's an example of a user-defined method in Java:

```
public class MyClass
{ static void myMethod() { System.out.println("Hello World!"); }
public static void main(String[] args) { myMethod(); // call the method } }
```

In the above example, we define a method called **myMethod** that simply prints "Hello World!" to the console when called. We then call the method in the **main** method using the **myMethod()** statement.
Similarly, a user-defined class in Java is a template for creating objects that define the properties and methods of that class. Here's an example:
csharp

```
public class MyClass {
 int x;
 public MyClass(int x) { this.x = x; }
public void printX() { System.out.println(x); }
public static void main(String[] args)
{ MyClass obj = new MyClass(5);// create an object of MyClass
obj.printX(); // call the printX method on the object } }
```

In the above example, we define a class called **MyClass** that has an integer property **x** and a method called **printX** that prints the value of **x** to the console. We then create an object of **MyClass** in the **main** method and call the **printX** method on that object using the **obj.printX()** statement.

Array
In Java, an array is a collection of similar data type elements stored in a contiguous memory location. Arrays can be of any data type, including primitive types and reference types.
Here is an example of creating and initializing an array of integers in Java:
javaCopy code
// declaring an array of integers int[] myArray; // allocating memory for 5 integers myArray = new int[5]; // initializing array elements myArray[0] = 1; myArray[1] = 2; myArray[2] = 3; myArray[3] = 4; myArray[4] = 5;
This creates an array **myArray** of length 5 and initializes its elements with values 1, 2, 3, 4, and 5. The elements of an array can be accessed using their index values, starting from 0. For example, **myArray[0]** returns the first element of the array, which is 1.

Chapter 2

Basic concept of OOP
Object-oriented programming (OOP) is a programming paradigm that is based on the concept of "objects", which can contain data and code that operate on that data. Java is an object-oriented programming language that fully supports the principles of OOP.
The key concepts in OOP are:

1. Classes: A class is a blueprint or a template for creating objects. It defines the attributes (data) and methods (functions) that the objects of that class will have.
2. Objects: An object is an instance of a class. It has its own set of data and methods that are defined by its class.
3. Encapsulation: This is the concept of grouping data and methods together into a single unit (class) and restricting access to the data from outside the class. In Java, this is achieved using access modifiers such as public, private, and protected.
4. Inheritance: This is the concept of creating a new class by inheriting the attributes and methods of an existing class. The new class is called a subclass or derived class, and the existing class is called the superclass or base class.
5. Polymorphism: This is the concept of using a single name or method to represent different implementations. In Java, this is achieved through method overloading and method overriding.

OOP in Java provides several benefits, including code reusability, modularity, and maintainability. By creating well-designed classes and objects, developers can create complex applications that are easier to understand and modify over time.

Code reusability
Code reusability is the ability to reuse existing code in a new context or project without having to rewrite it from scratch. It is an important concept in software development as it helps to save time, effort, and resources.

One of the best ways to achieve code reusability in Java is through the use of classes and inheritance. When you create a class, you define a blueprint for objects of that type. You can then create multiple instances of that class, each with its own set of data and behavior. By creating a class with reusable methods and data, you can use it in multiple projects without having to rewrite the same code again.
For example, let's say you are developing a program that needs to perform some common mathematical operations like addition, subtraction, multiplication, and division. You can create a MathUtils class that contains methods for these operations. Here is an example of such a class:

```
public class MathUtils {
  public static int add(int a, int b) {
    return a + b;
  }

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

  public static int multiply(int a, int b) {
    return a * b;
  }

  public static double divide(double a, double b) {
    if (b == 0) {
      throw new IllegalArgumentException("Cannot divide by zero!");
    }
    return a / b;
  }
}

import com.example.MathUtils;

public class MyProgram {
  public static void main(String[] args) {
    int result = MathUtils.add(5, 7);
    System.out.println("5 + 7 = " + result);
  }
}
```

Packages In Java
In Java, a package is a mechanism for organizing related classes, interfaces, and sub-packages. It provides a way to group related classes and interfaces into a single unit, making it easier to manage and maintain code. Packages help to avoid naming conflicts and provide a better structure to the program.
In a Java program, a package is declared using the "package" keyword followed by the name of the package. For example, if you want to create a package named "com.example.myproject", you would include the following statement at the beginning of your source code:

```
package com.example.myproject;
```

Classes and interfaces within a package can be accessed by other classes and interfaces within the same package without any special access modifiers. However, classes and interfaces in different packages need to be imported using the "import" keyword in order to use them.

```
import com.example.myproject.MyClass;

public class AnotherClass {
   public static void main(String[] args) {
      MyClass obj = new MyClass();
      // use obj here
   }
}
```

Packages provide a way to organize large projects into manageable units, and they also promote code reuse by making it easy to share classes and interfaces between different parts of a project or even between different projects.

Building User Defined Classes Package

Inheritance

Inheritance is a mechanism in Java where one class (called the subclass or derived class) can inherit the properties and behavior of another class (called the superclass or base class). The subclass can then add its own properties and behavior, and/or override the properties and behavior inherited from the superclass. Here's an example of inheritance in Java:

```
public class Animal {
   private String name;
   private int age;

   public Animal(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public void speak() {
      System.out.println("I am an animal.");
   }

   public String getName() {
      return name;
   }

   public int getAge() {
      return age;
   }
}

public class Dog extends Animal {
   private String breed;
```

```
  public Dog(String name, int age, String breed) {
    super(name, age);
    this.breed = breed;
  }

  @Override
  public void speak() {
    System.out.println("I am a dog.");
  }

  public String getBreed() {
    return breed;
  }
}

public class InheritanceExample {
  public static void main(String[] args) {
    Animal animal = new Animal("Generic Animal", 10);
    System.out.println("Name: " + animal.getName());
    System.out.println("Age: " + animal.getAge());
    animal.speak();

    Dog dog = new Dog("Fido", 3, "Labrador");
    System.out.println("Name: " + dog.getName());
    System.out.println("Age: " + dog.getAge());
    System.out.println("Breed: " + dog.getBreed());
    dog.speak();
  }
}
```

In Java, there are four types of inheritance:

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance (achieved through interfaces)

Here are examples of each type of inheritance:

1. Single Inheritance

Single inheritance is when a class inherits from only one parent class. In the example below, the **Dog** class extends the **Animal** class, which is its parent class. This means that the **Dog** class inherits all the properties and methods of the **Animal** class.

```
class Animal {
  public void eat() {
    System.out.println("Animal is eating");
  }
}

class Dog extends Animal {
  public void bark() {
    System.out.println("Dog is barking");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog myDog = new Dog();
    myDog.eat(); // Output: "Animal is eating"
    myDog.bark(); // Output: "Dog is barking"
  }
}
```

2.Multilevel inheritance is when a class inherits from a parent class, which in turn inherits from another parent class. In the example below, the **BabyDog** class extends the **Dog** class, which in turn extends the **Animal** class. This means that the **BabyDog** class inherits all the properties and methods of both the **Dog** class and the **Animal** class.

```
class Animal {
  public void eat() {
    System.out.println("Animal is eating");
  }
}

class Dog extends Animal {
  public void bark() {
    System.out.println("Dog is barking");
  }
}

class BabyDog extends Dog {
  public void weep() {
    System.out.println("Baby dog is weeping");
  }
}

public class Main {
  public static void main(String[] args) {
    BabyDog myBabyDog = new BabyDog();
    myBabyDog.eat(); // Output: "Animal is eating"
    myBabyDog.bark(); // Output: "Dog is barking"
    myBabyDog.weep(); // Output: "Baby dog is weeping"
  }
}
```

3. Hierarchical Inheritance
   Hierarchical inheritance is when one class is inherited by multiple child classes. In the example below, the **Cat** and **Dog** classes both inherit from the **Animal** class. This means that both the **Cat** and **Dog** classes inherit all the properties and methods of the **Animal** class.

```
class Animal {
  public void eat() {
    System.out.println("Animal is eating");
  }
}

class Dog extends Animal {
  public void bark() {
    System.out.println("Dog is barking");
  }
}

class Cat extends Animal {
  public void meow() {
    System.out.println("Cat is meowing");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog myDog = new Dog();
    myDog.eat(); // Output: "Animal is eating"
    myDog.bark(); // Output: "Dog is barking"

    Cat myCat = new Cat();
    myCat.eat(); // Output: "Animal is eating"
    myCat.meow(); // Output: "Cat is meowing"
  }
}
```

access protection Mechanism in java

In Java, access protection mechanisms are used to restrict access to class members (fields, methods, and inner classes) from outside the class, in order to encapsulate implementation details and improve code maintainability and security.

Java provides four levels of access protection, which are:

1. Private: Members declared as **private** can only be accessed within the same class. This ensures that the member is not accessible from outside the class, and can only be accessed through public methods of the class.
2. Default: Members declared with no access modifier (i.e. no **public**, **private**, or **protected** keyword) are called default or package-private members. They can only be accessed by classes in the same package. This is useful for classes that need to share implementation details with other classes in the same package.

3. Protected: Members declared as **protected** can be accessed by subclasses of the class, as well as by other classes in the same package. This allows for more flexible access control than the default level, while still maintaining encapsulation.
4. Public: Members declared as **public** can be accessed from any class in any package. This is the most permissive access level, and should be used sparingly to avoid exposing implementation details unnecessarily.
   In addition to these access modifiers, Java also provides the concept of nested classes, which can have different access levels depending on their declaration. For example, an inner class declared as **private** can only be accessed within the outer class, while an inner class declared as **public** can be accessed from any class.
   By using access protection mechanisms effectively, Java programmers can create classes that are more maintainable, extensible, and secure. Encapsulation is an important principle of object-oriented programming, and access modifiers allow developers to enforce it.

```java
class Ac{
  protected int a;
  private int x=10;

  public int getA() {
    return a;
  }

  public int getX() {
    return x;
  }

  public void show(){
    System.out.println(a);
  }
}

public class AccessModifiers {

//   public can be accessed from anywhere
//   private can only be accessed within the class not even on the same package
//   if we do not provide any access specifier then it is default,default simply means we can
access it within the same package
//   protected can be used in same class subclass different package subclass,but not in different
package non subclass

  public static void main(String[] args) {
    Ac ac = new Ac();
    ac.a=10;
    System.out.println(ac.getX()); ;
    ac.show();
    }
}
```