MCA-20202 Object Oriented Programming through JAVA

Instruction: 4 Periods/week **Time: 3 Hours** **Credits: 4**
Internal: 25 Marks **External: 75 Marks** **Total:100 Marks**

UNIT I

Introduction to OOP: Introduction, Principles of Object Oriented Languages, Applications of OOP, Programming Constructs: Variables, Primitive Datatypes, Identifiers- Naming Conventions, Keywords, Literals, Operators-Binary, Unary and ternary, Expressions, Precedence rules and Associativity, Primitive Type Conversion and Casting, Flow of control-Branching, Conditional, loops. Classes and Objects- classes, Objects, Creating Objects, Methods, constructors-Constructor overloading, cleaning up unused objects-Garbage collector, Class variable and Methods-Static keyword, this keyword, Arrays, Command line arguments.

Inheritance: Types of Inheritance, Deriving classes using extends keyword, Method overloading, super keyword, final keyword, Abstract class.

UNIT II

Interfaces, Packages and Enumeration: Interface-Extending interface, Interface Vs Abstract classes, Packages-Creating packages, using Packages, Access protection, java.lang package.

Exceptions & Assertions – Introduction, Exception handling techniques- try… catch, throw, throws, finally block, user defined exception, Exception Encapsulation and Enrichment, Assertions.

UNIT III

MultiThreading: java.lang.Thread, The main Thread, Creation of new threads, Thread priority, Multithreading- Using isAlive () and join (), Synchronization, suspending and Resuming threads, Communication between Threads Input/Output: reading and writing data, java.io package, **Applets**– Applet class, Applet structure, An Example Applet Program, Applet : Life Cycle, paint(), update() and repaint().

UNIT IV

Event Handling -Introduction, Event Delegation Model, java.awt.event Description, Sources of Events, Event Listeners, Adapter classes, Inner classes.

Abstract Window Toolkit:Why AWT?, java.awt package, Components and Containers, Button, Label, Checkbox, Radio buttons, List boxes, Choice boxes, Text field and Text area, container classes, Layouts, Menu, Scroll bar, **Swing**: Introduction, JFrame, JApplet, JPanel, Components in swings, Layout Managers, JList and JScroll Pane, Split Pane, JTabbedPane, Dialog Box Pluggable Look and Feel.

Text Books:

1. The Complete Reference Java, 8ed, Herbert Schildt, TMH
2. Programming in JAVA, Sachin Malhotra, Saurabhchoudhary, Oxford.

References:

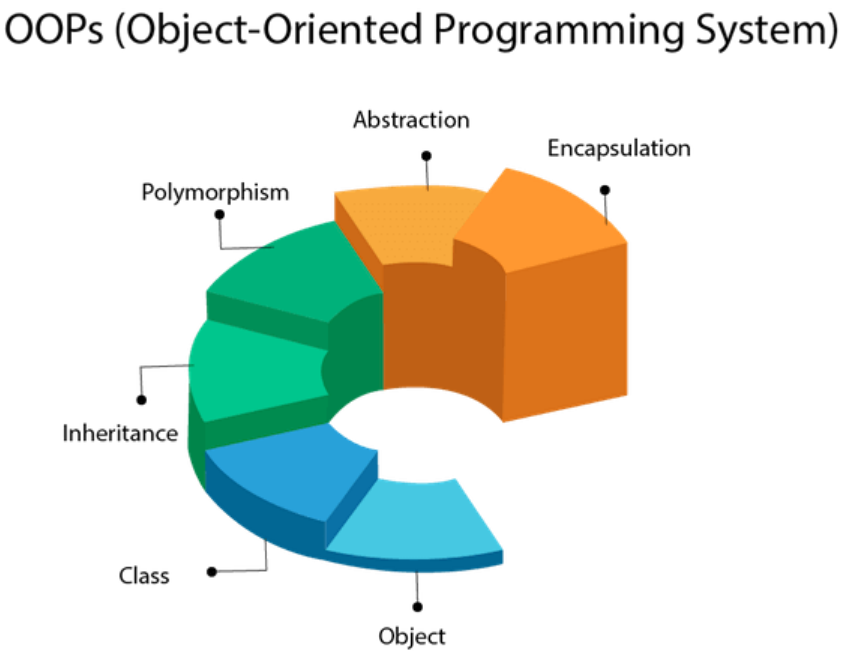
1. JAVA for Beginners, 4e, Joyce Farrell, Ankit R. Bhavsar, Cengage Learning.
2. Introduction to Java programming, 7th ed, Y Daniel Liang, Pearson.

UNIT-I

OOPs (Object-Oriented Programming System)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation



Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

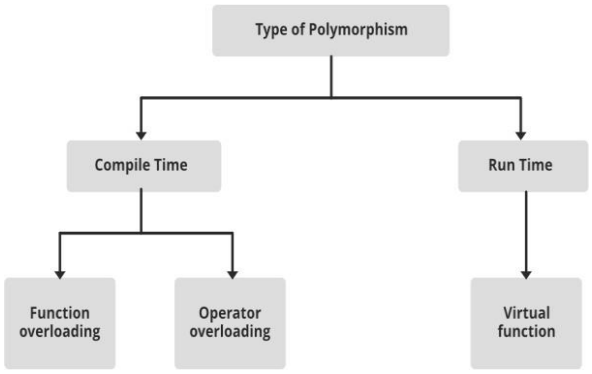
Collection of objects is called class. It is a logical entity.
A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.



In Java, we use method overloading and method overriding to achieve polymorphism.
Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.
In Java, we use **abstract class and interface** to achieve abstraction.

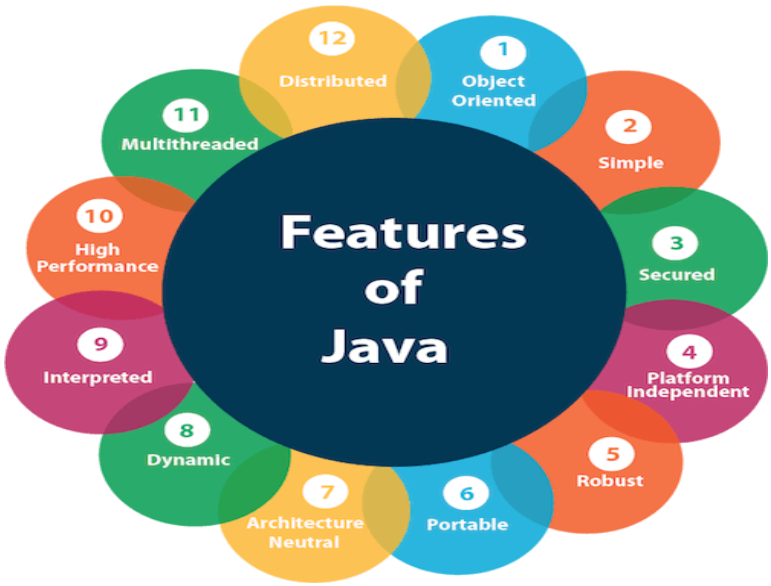
Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.
A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

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



1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

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.

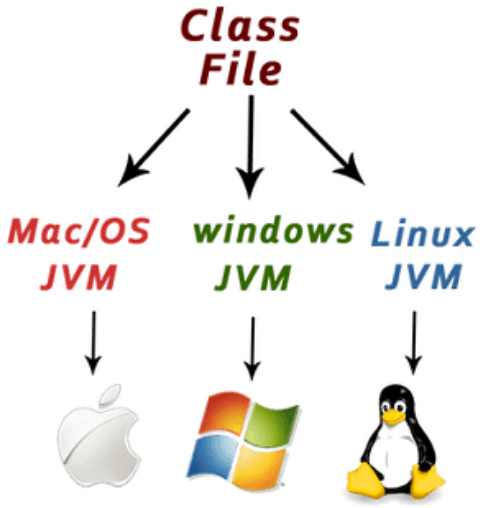
Skip Ad

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

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. 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:

1. Runtime Environment
2. API(Application Programming Interface)

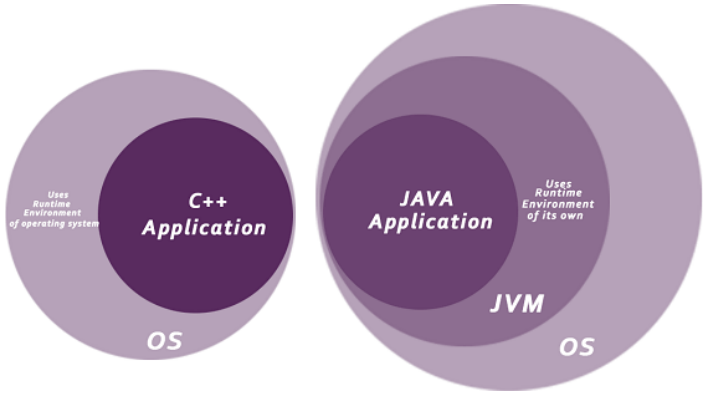
Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

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**



- **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

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).

Java Variables

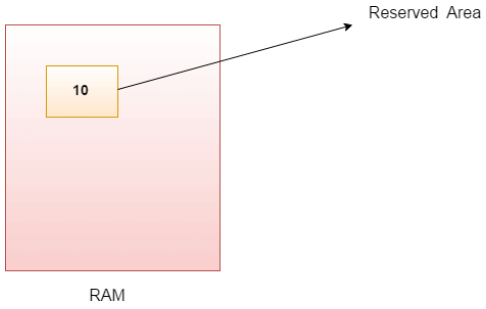
A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

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

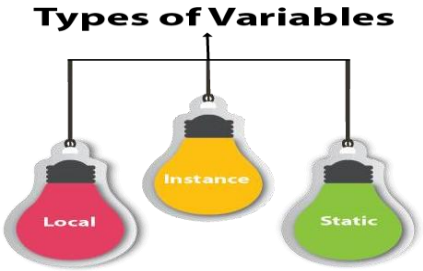


1. **int** data=50;//Here data is variable

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

```
1.  public class A
2.  {
3.     static int m=100;//static variable
4.     void method()
5.     {
6.        int n=90;//local variable
7.     }
8.     public static void main(String args[])
9.     {
10.       int data=50;//instance variable
```

```
11.     }
12. }//end of class
10.0
```

Java Variable Example: Narrowing (Typecasting)

```
1. public class Simple{
2. public static void main(String[] args){
3. float f=10.5f;
4. //int a=f;//Compile time error
5. int a=(int)f;
6. System.out.println(f);
7. System.out.println(a);
8. }}
```

Output:

```
10.5
10
```

Java Variable Example: Overflow

```
1. class Simple{
2. public static void main(String[] args){
3. //Overflow
4. int a=130;
5. byte b=(byte)a;
6. System.out.println(a);
7. System.out.println(b);
8. }}
```

Output:

```
130
-126
```

Java Variable Example: Adding Lower Type

```
1. class Simple{
2. public static void main(String[] args){
3. byte a=10;
4. byte b=10;
5. //byte c=a+b;//Compile Time Error: because a+b=20 will be int
6. byte c=(byte)(a+b);
7. System.out.println(c);
8. }}
```

Output:

```
20
```

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.

Java Primitive Data Types

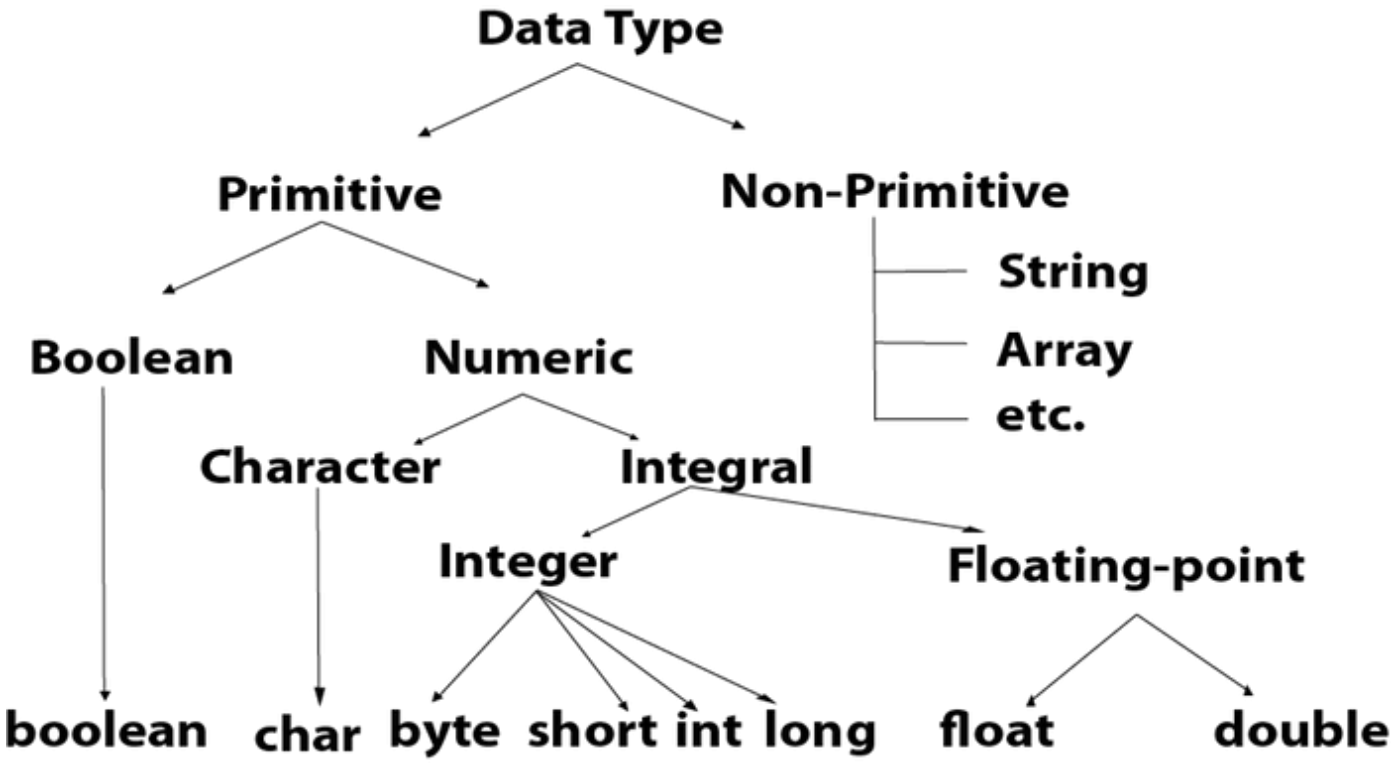
In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

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'

Operators in Java

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

There are many types of operators in Java which are given below:

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

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^

	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java 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

Java Unary Operator Example: ++ and --

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int x=10;
4. System.out.println(x++);//10 (11)
5. System.out.println(++x);//12
6. System.out.println(x--);//12 (11)
7. System.out.println(--x);//10
8. }}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=10;
5. System.out.println(a++ + ++a);//10+12=22
6. System.out.println(b++ + b++);//10+11=21
7.
8. }}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=-10;
5. boolean c=true;
6. boolean d=false;
7. System.out.println(~a);//-11 (minus of total positive value which starts from 0)
8. System.out.println(~b);//9 (positive of total minus, positive starts from 0)
9. System.out.println(!c);//false (opposite of boolean value)
10. System.out.println(!d);//true
11. }}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operators

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

Java Arithmetic Operator Example

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

Output:

```
15
5
50
2
0
```

Java Arithmetic Operator Example: Expression

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10*10/5+3-1*4/2);
4. }}
```

Output:

```
21
```

Java 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.

Java Left Shift Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10<<2);//10*2^2=10*4=40
4. System.out.println(10<<3);//10*2^3=10*8=80
5. System.out.println(20<<2);//20*2^2=20*4=80
6. System.out.println(15<<4);//15*2^4=15*16=240
7. }}
```

Output:

```
40
80
80
240
```

Java Right Shift Operator

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

Java Right Shift Operator Example

```
1. public OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10>>2);//10/2^2=10/4=2
4. System.out.println(20>>2);//20/2^2=20/4=5
5. System.out.println(20>>3);//20/2^3=20/8=2
6. }}
```

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>>

```
1. public class OperatorExample{
2. public static void main(String args[]){
3.     //For positive number, >> and >>> works same
4.     System.out.println(20>>2);
5.     System.out.println(20>>>2);
6.     //For negative number, >>> changes parity bit (MSB) to 0
7.     System.out.println(-20>>2);
8.     System.out.println(-20>>>2);
9. }}
```

Output:

```
5
5
-5
1073741819
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a<b&&a<c);//false && true = false
7. System.out.println(a<b&a<c);//false & true = false
8. }}
```

Output:

```
false
```

```
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a<b&&a++<c);//false && true = false
7. System.out.println(a);//10 because second condition is not checked
8. System.out.println(a<b&a++<c);//false && true = false
9. System.out.println(a);//11 because second condition is checked
10. }}
```

Output:

```
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a>b||a<c);//true || true = true
7. System.out.println(a>b|a<c);//true | true = true
8. //|| vs |
```

```
9. System.out.println(a>b||a++<c);//true || true = true
10. System.out.println(a);//10 because second condition is not checked
11. System.out.println(a>b|a++<c);//true | true = true
12. System.out.println(a);//11 because second condition is checked
13. }}
```

Output:

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

Java Ternary Operator

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

Java Ternary Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=2;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Output:

```
2
```

Another Example:

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
```

```
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Output:

```
5
```

Java Assignment Operator

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

Java Assignment Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=20;
5. a+=4;//a=a+4 (a=10+4)
6. b-=4;//b=b-4 (b=20-4)
7. System.out.println(a);
8. System.out.println(b);
9. }}
```

Output:

```
14
16
```

Java Assignment Operator Example

```
1. public class OperatorExample{
2. public static void main(String[] args){
3. int a=10;
4. a+=3;//10+3
5. System.out.println(a);
6. a-=4;//13-4
7. System.out.println(a);
```

```
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

Output:

```
13
9
18
9
```

Java Assignment Operator Example: Adding short

```
public class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
//a+=b;//a=a+b internally so fine
a=a+b;//Compile time error because 10+10=20 now int
System.out.println(a);
}}
```

Output:

```
Compile time error
```

After type cast:

```
public class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
a=(short)(a+b);//20 which is int now converted to short
System.out.println(a);
}}
```

Output:

```
20
```

Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

List of Java Keywords

A list of Java keywords or reserved words are given below:

1. **abstract:** Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.
2. **boolean:** Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. **break:** Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. **byte:** Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. **case:** Java case keyword is used with the switch statements to mark blocks of text.
6. **catch:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. **char:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. **class:** Java class keyword is used to declare a class.
9. **continue:** Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. **default:** Java default keyword is used to specify the default block of code in a switch statement.
11. **do:** Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.

12. **double:** Java double keyword is used to declare a variable that can hold 64-bit floating-point number.
13. **else:** Java else keyword is used to indicate the alternative branches in an if statement.
14. **enum:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. **extends:** Java extends keyword is used to indicate that a class is derived from another class or interface.
16. **final:** Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.
17. **finally:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.
18. **float:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. **for:** Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.
20. **if:** Java if keyword tests the condition. It executes the if block if the condition is true.
21. **implements:** Java implements keyword is used to implement an interface.
22. **import:** Java import keyword makes classes and interfaces available and accessible to the current source code.
23. **instanceof:** Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. **int:** Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. **interface:** Java interface keyword is used to declare an interface. It can have only abstract methods.
26. **long:** Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native:** Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. **new:** Java new keyword is used to create new objects.
29. **null:** Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.

30. **package:** Java package keyword is used to declare a Java package that includes the classes.
31. **private:** Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. **protected:** Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.
33. **public:** Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. **return:** Java return keyword is used to return from a method when its execution is complete.
35. **short:** Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. **static:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
37. **strictfp:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. **super:** Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
39. **switch:** The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. **synchronized:** Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. **this:** Java this keyword can be used to refer the current object in a method or constructor.
42. **throw:** The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.
43. **throws:** The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.
44. **transient:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. **try:** Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. **void:** Java void keyword is used to specify that a method does not have a return value.

47. **volatile:** Java volatile keyword is used to indicate that a variable may change asynchronously.
48. **while:** Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use the while loop.

Java Naming Convention

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of Naming Conventions in Java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

Naming Conventions of the Different Identifiers

The following table shows the popular conventions used for the different identifiers.

Identifiers Type	Naming Rules	Examples

Class	It should start with the uppercase letter.
	It should be a noun such as Color, Button, System, Thread, etc.
	Use appropriate words, instead of acronyms.
		public class **Employee**
		{
		//code snippet
		}

Interface	It should start with the uppercase letter.
	It should be an adjective such as Runnable, Remote, ActionListener.
		interface **Printable**
		{
		//code snippet

	Use appropriate words, instead of acronyms.	}

Method	It should start with lowercase letter.
	It should be a verb such as main(), print(), println().
	If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
		class Employee
		{
		// method
		void **draw()**
		{
		//code snippet
		}
		}

Variable	It should start with a lowercase letter such as id, name.
	It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
	If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
	Avoid using one-character variables such as x, y, z.
		class Employee
		{
		// variable
		int **id**;
		//code snippet
		}

Package	It should be a lowercase letter such as java, lang.
	If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
		//package
		package **com.javatpoint;**
		class Employee
		{
		//code snippet
		}

Constant	It should be in uppercase letters such as RED, YELLOW.
	If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
	It may contain digits but not as the first letter.
		class Employee
		{
		//constant
		static final
		int **MIN_AGE** = 18;
		//code snippet
		}

CamelCase in Java naming conventions

Java follows camel-case syntax for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

Associativity of Operators in Java

A Java operator is a special symbol that performs a certain operation on multiple operands and gives the result as an output.

Java has a large number of operators that are divided into two categories. First, an operator's performance is based on the number of operands it performs on. Second, the type or nature of the operation performed by an operator.

Operators can be categorized into the following groups based on the sort of operation they perform:

1. Arithmetic Operators
2. Increment Decrement Operators
3. Assignment Operators
4. Bitwise Operators
5. Relational Operators
6. Logical Operators
7. Miscellaneous Operators

Java Operators Precedence and Associativity

Precedence and associativity are two features of Java operators. When there are two or more operators in an expression, the operator with the highest priority will be executed first.

For example, consider the equation, 1 + 2 * 5. Here, the multiplication (*) operator is executed first, followed by addition. Because multiplication operator takes precedence over the addition operator.

Alternatively, when an operand is shared by two operators (2 in the example above is shared by + and *), the higher priority operator processes the shared operand. You should have grasped the significance of precedence or priority in the execution of operators from the preceding example.

However, the situation may not always be as obvious as in the example above. What if the precedence of all operators in an expression is the same? In that instance, the second quality associated with an operator, associativity, comes into existence.

Associativity specifies the order in which operators are executed, which can be left to right or right to left. For example, in the phrase a = b = c = 8, the assignment operator is used from right

to left. It means that the value 8 is assigned to c, then c is assigned to b, and at last b is assigned to a. This phrase can be parenthesized as (a = (b = (c = 8)).

The priority of a Java operator can be modified by putting parenthesis around the lower order priority operator, but not the associativity. In the equation (1 + 2) * 3, for example, the addition will be performed first since parentheses take precedence over the multiplication operator.

Operator Precedence in Java (Highest to Lowest)

Category	Operators	Associativity
Postfix	++ - -	Left to right
Unary	+ - ! ~ ++ - -	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right

Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left

Java Operator Associativity

Operators with the same precedence follow the operator group's operator associativity. Operators in Java can be left-associative, right-associative, or have no associativity at all. Left-associative operators are assessed from left to right, right-associative operators are reviewed from right to left, and operators with no associativity are evaluated in any order.

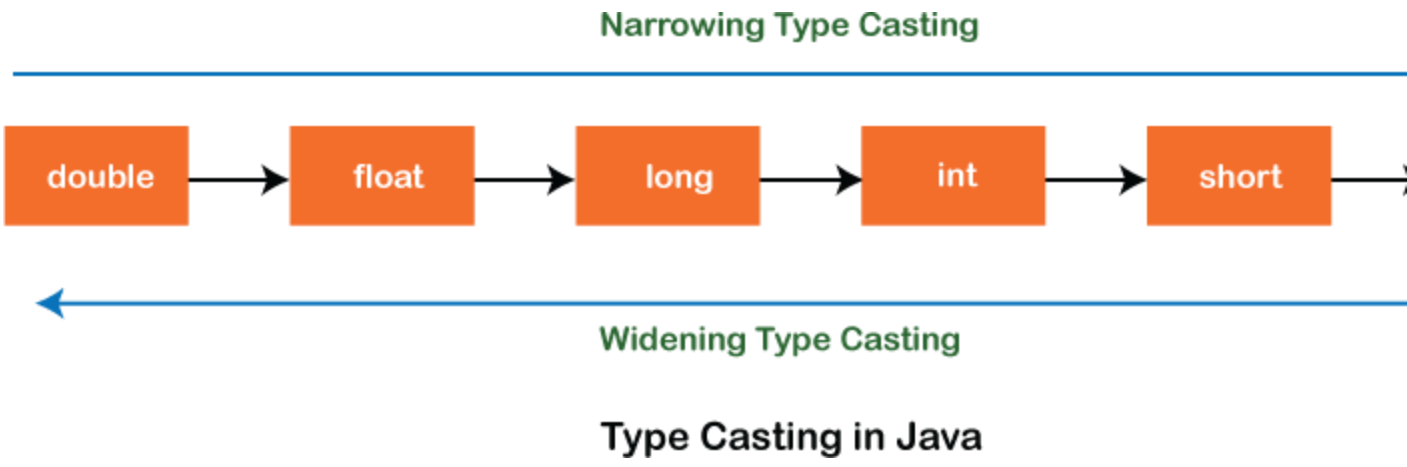
Operator Precedence Vs. Operator Associativity

The operator's **precedence** refers to the order in which operators are evaluated within an expression whereas **associativity** refers to the order in which the consecutive operators within the same group are carried out.

Precedence rules specify the priority (which operators will be evaluated first) of operators.

Type Casting in Java

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and **its types** with proper examples.



Type casting

Convert a value from one data type to another data type is known as **type casting**.

Types of Type Casting

There are two types of type casting:

- Widening Type Casting
- Narrowing Type Casting

Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

1. **byte -> short -> char -> int -> long -> float -> double**

   For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

   WideningTypeCastingExample.java

```
1. public class WideningTypeCastingExample
2. {
3. public static void main(String[] args)
4. {
5. int x = 7;
6. //automatically converts the integer type into long type
7. long y = x;
8. //automatically converts the long type into float type
9. float z = y;
10. System.out.println("Before conversion, int value "+x);
11. System.out.println("After conversion, long value "+y);
12. System.out.println("After conversion, float value "+z);
13. }
```

```
14. }
```

Output

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

1. **double** -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

Let's see an example of narrowing type casting.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
1. public class NarrowingTypeCastingExample
2. {
3. public static void main(String args[])
4. {
5. double d = 166.66;
6. //converting double data type into long data type
7. long l = (long)d;
8. //converting long data type into int data type
9. int i = (int)l;
10. System.out.println("Before conversion: "+d);
11. //fractional part lost
12. System.out.println("After conversion into long type: "+l);
13. //fractional part lost
```

```
14. System.out.println("After conversion into int type: "+i);
15. }
16. }
```

Output

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

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.

```
1. if(condition) {
2. statement 1; //executes when condition is true
3. }
```

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

Student.java

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. int x = 10;
4. int y = 12;
5. if(x+y > 20) {
6. System.out.println("x + y is greater than 20");
7. }
8. }
9. }
```

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:

```
1. if(condition) {
2. statement 1; //executes when condition is true
3. }
4. else{
5. statement 2; //executes when condition is false
6. }
```

Consider the following example.

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. int x = 10;
4. int y = 12;
5. if(x+y < 10) {
6. System.out.println("x + y is less than 10");
7. } else {
8. System.out.println("x + y is greater than 20");
9. }
10. }
11. }
```

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.

```
1. if(condition 1) {
2. statement 1; //executes when condition 1 is true
3. }
4. else if(condition 2) {
5. statement 2; //executes when condition 2 is true
6. }
7. else {
8. statement 2; //executes when all the conditions are false
9. }
```

Consider the following example.

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. String city = "Delhi";
4. if(city == "Meerut") {
5. System.out.println("city is meerut");
6. }else if (city == "Noida") {
7. System.out.println("city is noida");
8. }else if(city == "Agra") {
9. System.out.println("city is agra");
10. }else {
11. System.out.println(city);
12. }
13. }
14. }
```

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.

```
1. if(condition 1) {
2. statement 1; //executes when condition 1 is true
3. if(condition 2) {
4. statement 2; //executes when condition 2 is true
5. }
6. else{
7. statement 2; //executes when condition 2 is false
8. }
9. }
```

Consider the following example.

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. String address = "Delhi, India";
4.
5. if(address.endsWith("India")) {
6. if(address.contains("Meerut")) {
7. System.out.println("Your city is Meerut");
8. }else if(address.contains("Noida")) {
9. System.out.println("Your city is Noida");
10. }else {
11. System.out.println(address.split(",")[0]);
12. }
13. }else {
14. System.out.println("You are not living in India");
15. }
16. }
```

```
17. }
```

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.

```
1. switch (expression){
2.     case value1:
3.      statement1;
4.      break;
5.     .
6.     .
7.     .
8.     case valueN:
9.      statementN;
10.     break;
11.    default:
```

```
12.     default statement;
13. }
```

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

Student.java

```
1. public class Student implements Cloneable {
2. public static void main(String[] args) {
3. int num = 2;
4. switch (num){
5. case 0:
6. System.out.println("number is 0");
7. break;
8. case 1:
9. System.out.println("number is 1");
10. break;
11. default:
12. System.out.println(num);
13. }
14. }
15. }
```

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

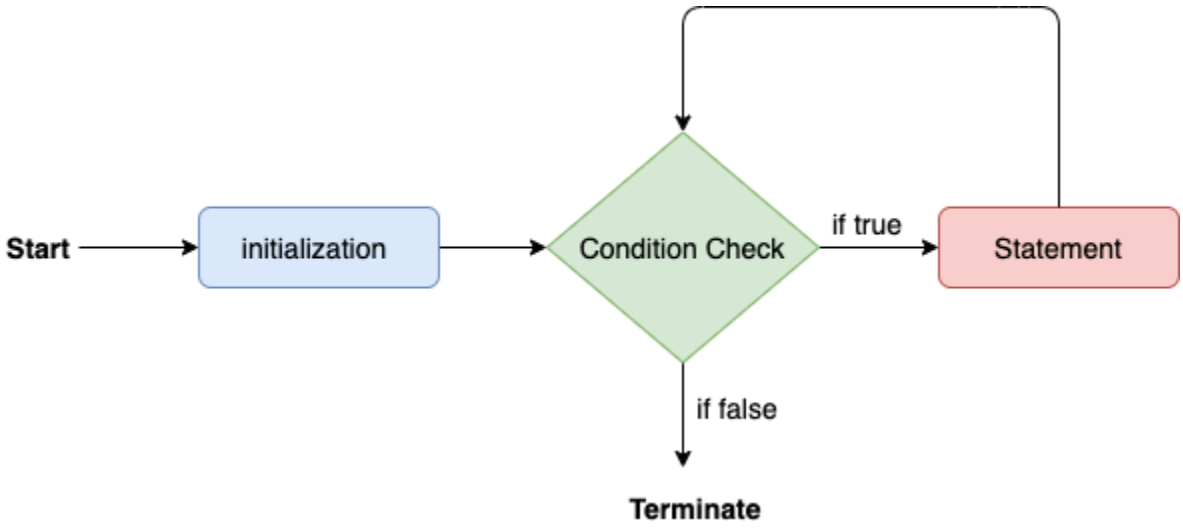
Let's understand the loop statements one by one.

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.

```
1. for(initialization, condition, increment/decrement) {
2. //block of statements
3. }
```

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

```
1. public class Calculattion {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. int sum = 0;
```

```
5. for(int j = 1; j<=10; j++) {
6. sum = sum + j;
7. }
8. System.out.println("The sum of first 10 natural numbers is " + sum);
9. }
10. }
```

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.

```
1. for(data_type var : array_name/collection_name){
2. //statements
3. }
```

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

Calculation.java

```
1. public class Calculation {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. String[] names = {"Java","C","C++","Python","JavaScript"};
5. System.out.println("Printing the content of the array names:\n");
6. for(String name:names) {
7. System.out.println(name);
8. }
9. }
10. }
```

Java while loop

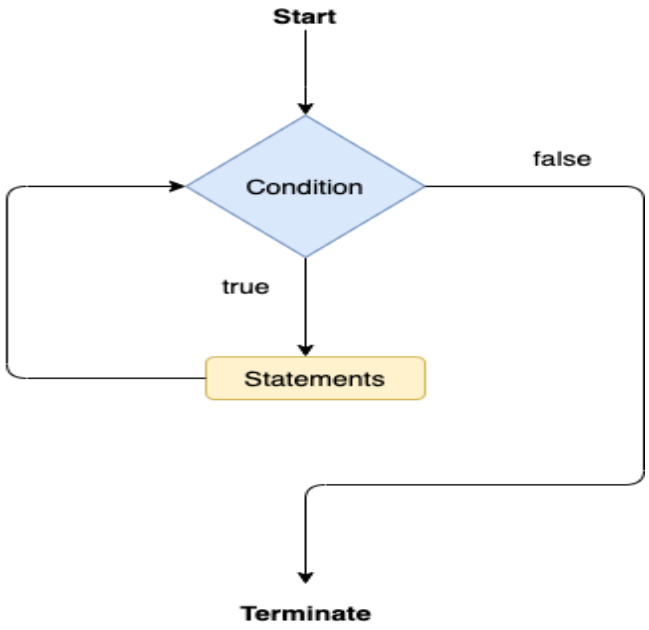
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while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

```
1. while(condition){
2. //looping statements
3. }
```

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



Consider the following example.

Calculation .java

```
1. public class Calculation {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
```

```
4. int i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. while(i<=10) {
7. System.out.println(i);
8. i = i + 2;
9. }
10. }
11. }
```

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

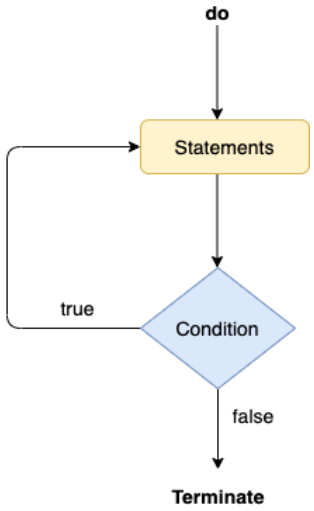
Java do-while loop

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.

```
1. do
2. {
3. //statements
4. } 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

```
1. public class Calculation {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. int i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. do {
7. System.out.println(i);
8. i = i + 2;
9. }while(i<=10);
10. }
11. }
```

Output:

```
Printing the list of first 10 even numbers
0
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

```
1. public class BreakExample {
2.
3. public static void main(String[] args) {
4. // TODO Auto-generated method stub
5. for(int i = 0; i<= 10; i++) {
6. System.out.println(i);
7. if(i==6) {
8. break;
9. }
10. }
11. }
12. }
```

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

break statement example with labeled for loop

Calculation.java

```
1. public class Calculation {
2.
3. public static void main(String[] args) {
4. // TODO Auto-generated method stub
5. a:
6. for(int i = 0; i<= 10; i++) {
7. b:
8. for(int j = 0; j<=15;j++) {
9. c:
10. for (int k = 0; k<=20; k++) {
11. System.out.println(k);
12. if(k==5) {
13. break a;
14. }
15. }
16. }
17.
18. }
19. }
20.
21.
22. }
```

Output:

```
0
1
```

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.

```
1.  public class ContinueExample {
2.
3.  public static void main(String[] args) {
4.  // TODO Auto-generated method stub
5.
6.  for(int i = 0; i<= 2; i++) {
7.
8.  for (int j = i; j<=5; j++) {
9.
10. if(j == 4) {
11. continue;
12. }
13. System.out.println(j);
14. }
15. }
16. }
17.
18. }
```

Output

```
3
5
```

Java If-else Statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in Java.

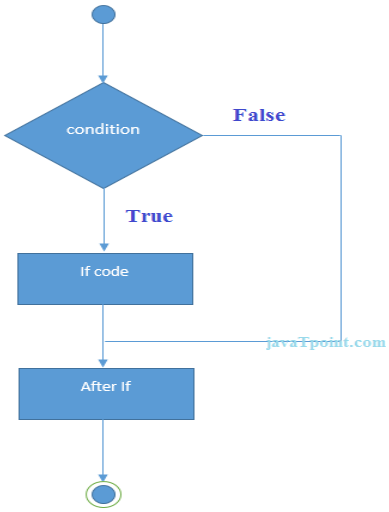
- if statement
- if-else statement
- if-else-if ladder
- nested if statement

Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

```
1. if(condition){
2. //code to be executed
3. }
```



Example:

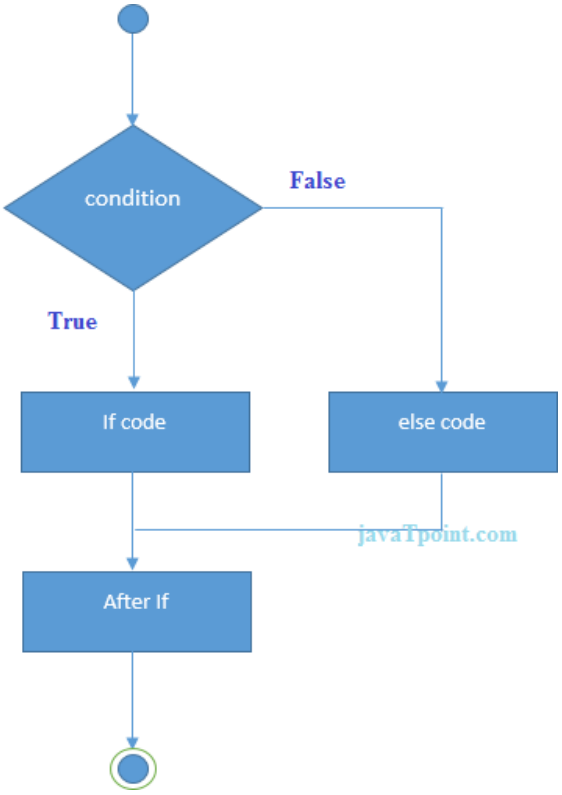
```
1. //Java Program to demonstate the use of if statement.
2. public class IfExample {
3. public static void main(String[] args) {
4.     //defining an 'age' variable
5.     int age=20;
6.     //checking the age
7.     if(age>18){
8.         System.out.print("Age is greater than 18");
9.     }
10. }
11. }
```

Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
1. if(condition){
2. //code if condition is true
3. }else{
4. //code if condition is false
5. }
```



Example:

```
1. //A Java Program to demonstrate the use of if-else statement.
2. //It is a program of odd and even number.
3. public class IfElseExample {
4. public static void main(String[] args) {
5.     //defining a variable
6.     int number=13;
7.     //Check if the number is divisible by 2 or not
8.     if(number%2==0){
```

```
9.          System.out.println("even number");
10.     }else{
11.         System.out.println("odd number");
12.     }
13. }
14. }
```

Output:

```
odd number
```

Leap Year Example:

A year is leap, if it is divisible by 4 and 400. But, not by 100.

```
1.  public class LeapYearExample {
2.  public static void main(String[] args) {
3.      int year=2020;
4.      if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
5.          System.out.println("LEAP YEAR");
6.      }
7.      else{
8.          System.out.println("COMMON YEAR");
9.      }
10. }
11. }
```

Output:

```
LEAP YEAR
```

Using Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

Example:

```
1.  public class IfElseTernaryExample {
```

```
2.  public static void main(String[] args) {
3.     int number=13;
4.     //Using ternary operator
5.     String output=(number%2==0)?"even number":"odd number";
6.     System.out.println(output);
7.  }
8.  }
```

Output:

```
odd number
```

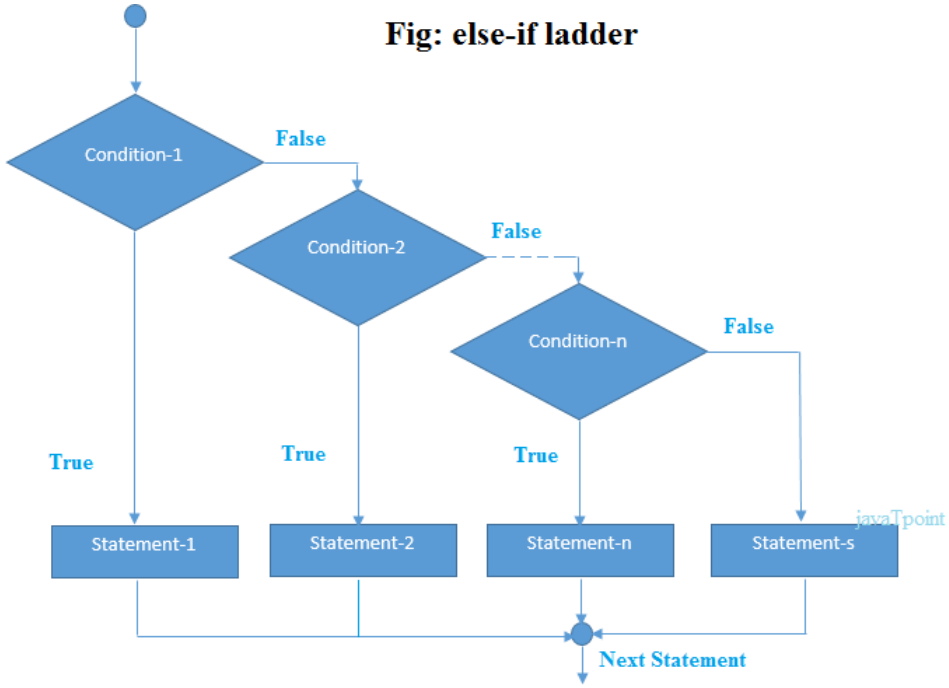
Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
1.  if(condition1){
2.  //code to be executed if condition1 is true
3.  }else if(condition2){
4.  //code to be executed if condition2 is true
5.  }
6.  else if(condition3){
7.  //code to be executed if condition3 is true
8.  }
9.  ...
10. else{
11. //code to be executed if all the conditions are false
12. }
```

Fig: else-if ladder



Example:

```
1.  //Java Program to demonstrate the use of If else-if ladder.
2.  //It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
3.  public class IfElseIfExample {
4.  public static void main(String[] args) {
5.      int marks=65;
6.
7.      if(marks<50){
8.          System.out.println("fail");
9.      }
10.     else if(marks>=50 && marks<60){
11.         System.out.println("D grade");
12.     }
13.     else if(marks>=60 && marks<70){
14.         System.out.println("C grade");
15.     }
16.     else if(marks>=70 && marks<80){
17.         System.out.println("B grade");
```

```
18.     }
19.     else if(marks>=80 && marks<90){
20.         System.out.println("A grade");
21.     }else if(marks>=90 && marks<100){
22.         System.out.println("A+ grade");
23.     }else{
24.         System.out.println("Invalid!");
25.     }
26. }
27. }
```

Output:

```
C grade
```

Program to check POSITIVE, NEGATIVE or ZERO:

```
1.  public class PositiveNegativeExample {
2.  public static void main(String[] args) {
3.      int number=-13;
4.      if(number>0){
5.      System.out.println("POSITIVE");
6.      }else if(number<0){
7.      System.out.println("NEGATIVE");
8.      }else{
9.      System.out.println("ZERO");
10.    }
11. }
12. }
```

Output:

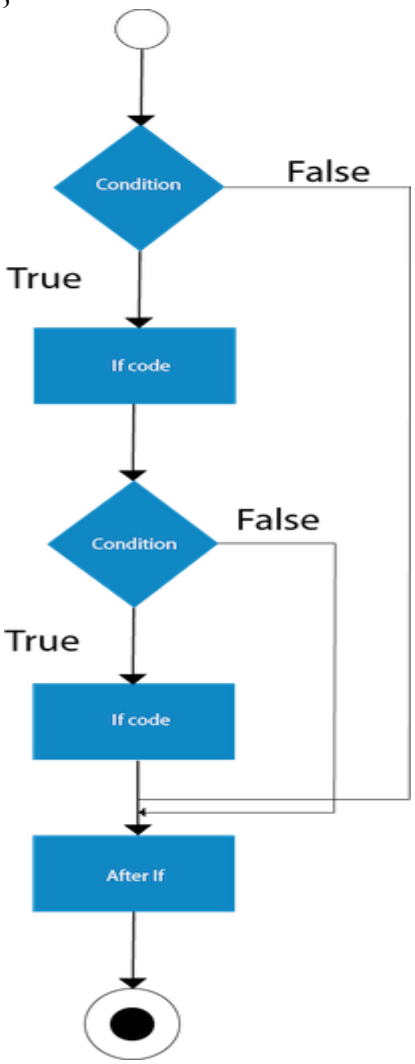
```
NEGATIVE
```

Java Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
1. if(condition){
2.     //code to be executed
3.         if(condition){
4.             //code to be executed
5.     }
6. }
```



Example:

```
1. //Java Program to demonstrate the use of Nested If Statement.
2. public class JavaNestedIfExample {
3. public static void main(String[] args) {
```

```
4.      //Creating two variables for age and weight
5.      int age=20;
6.      int weight=80;
7.      //applying condition on age and weight
8.      if(age>=18){
9.         if(weight>50){
10.           System.out.println("You are eligible to donate blood");
11.        }
12.     }
13. }}
```

Output:

```
You are eligible to donate blood
```

Example 2:

```
1.  //Java Program to demonstrate the use of Nested If Statement.
2.  public class JavaNestedIfExample2 {
3.  public static void main(String[] args) {
4.      //Creating two variables for age and weight
5.      int age=25;
6.      int weight=48;
7.      //applying condition on age and weight
8.      if(age>=18){
9.         if(weight>50){
10.           System.out.println("You are eligible to donate blood");
11.        } else{
12.           System.out.println("You are not eligible to donate blood");
13.        }
14.     } else{
15.       System.out.println("Age must be greater than 18");
16.     }
17. }  }
```

Output:

You are not eligible to donate blood

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

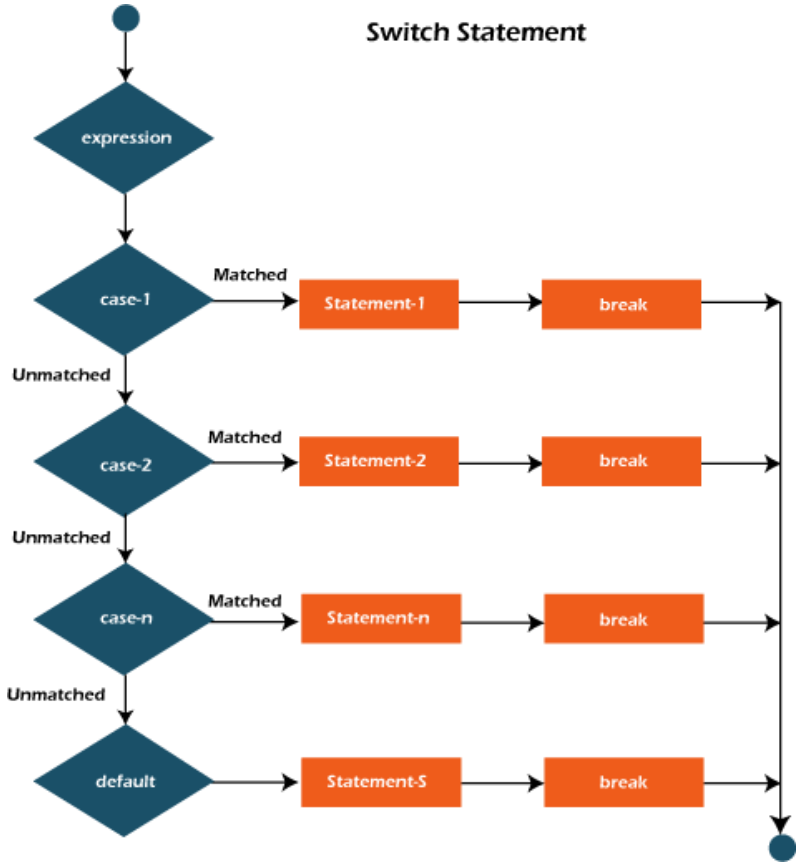
Points to Remember

- There can be *one or N number of case values* for a switch expression.
- The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of *byte, short, int, long (with its Wrapper type), enums and string*.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
- The case value can have a *default label* which is optional.

Syntax:

```
1. switch(expression){
2. case value1:
3.  //code to be executed;
4.  break;  //optional
5. case value2:
6.  //code to be executed;
7.  break;  //optional
8. ......
9.
10. default:
11.  code to be executed if all cases are not matched;
12. }
```

Flowchart of Switch Statement



Example:

SwitchExample.java

```
1.  public class SwitchExample {
2.  public static void main(String[] args) {
3.      //Declaring a variable for switch expression
4.      int number=20;
5.      //Switch expression
6.      switch(number){
7.      //Case statements
8.      case 10: System.out.println("10");
9.      break;
10.     case 20: System.out.println("20");
```

```
11.     break;
12.     case 30: System.out.println("30");
13.     break;
14.     //Default case statement
15.     default:System.out.println("Not in 10, 20 or 30");
16.     }
17. }
18. }
```

Output:

```
20
```

Finding Month Example:

SwitchMonthExample.javaHTML

```
1.  //Java Program to demonstrate the example of Switch statement
2.  //where we are printing month name for the given number
3.  public class SwitchMonthExample {
4.  public static void main(String[] args) {
5.      //Specifying month number
6.      int month=7;
7.      String monthString="";
8.      //Switch statement
9.      switch(month){
10.     //case statements within the switch block
11.     case 1: monthString="1 - January";
12.     break;
13.     case 2: monthString="2 - February";
14.     break;
15.     case 3: monthString="3 - March";
16.     break;
17.     case 4: monthString="4 - April";
18.     break;
19.     case 5: monthString="5 - May";
```

```
20.     break;
21.     case 6: monthString="6 - June";
22.     break;
23.     case 7: monthString="7 - July";
24.     break;
25.     case 8: monthString="8 - August";
26.     break;
27.     case 9: monthString="9 - September";
28.     break;
29.     case 10: monthString="10 - October";
30.     break;
31.     case 11: monthString="11 - November";
32.     break;
33.     case 12: monthString="12 - December";
34.     break;
35.     default:System.out.println("Invalid  Month!");
36.     }
37.     //Printing month of the given number
38.     System.out.println(monthString);
39. }
40. }
```

Output:

```
7 - July
```

Program to check Vowel or Consonant:

If the character is A, E, I, O, or U, it is vowel otherwise consonant. It is not case-sensitive.

SwitchVowelExample.java

```
1.  public class SwitchVowelExample {
2.  public static void main(String[] args) {
3.      char ch='O';
4.      switch(ch)
5.      {
```

```
6.          case 'a':
7.              System.out.println("Vowel");
8.              break;
9.          case 'e':
10.             System.out.println("Vowel");
11.             break;
12.         case 'i':
13.             System.out.println("Vowel");
14.             break;
15.         case 'o':
16.             System.out.println("Vowel");
17.             break;
18.         case 'u':
19.             System.out.println("Vowel");
20.             break;
21.         case 'A':
22.             System.out.println("Vowel");
23.             break;
24.         case 'E':
25.             System.out.println("Vowel");
26.             break;
27.         case 'I':
28.             System.out.println("Vowel");
29.             break;
30.         case 'O':
31.             System.out.println("Vowel");
32.             break;
33.         case 'U':
34.             System.out.println("Vowel");
35.             break;
36.         default:
37.             System.out.println("Consonant");
38.     }
39. }
```

```
40. }
```

Output:

```
Vowel
```

Java Switch Statement is fall-through

The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

Example:

SwitchExample2.java

```
1.  //Java Switch Example where we are omitting the
2.  //break statement
3.  public class SwitchExample2 {
4.  public static void main(String[] args) {
5.      int number=20;
6.      //switch expression with int value
7.      switch(number){
8.      //switch cases without break statements
9.      case 10: System.out.println("10");
10.     case 20: System.out.println("20");
11.     case 30: System.out.println("30");
12.     default:System.out.println("Not in 10, 20 or 30");
13.     }
14. }
15. }
```

Test it Now

Output:

```
20
30
Not in 10, 20 or 30
```

Java Switch Statement with String

Java allows us to use strings in switch expression since Java SE 7. The case statement should be string literal.

Example:

SwitchStringExample.java

```
1. //Java Program to demonstrate the use of Java Switch
2. //statement with String
3. public class SwitchStringExample {
4. public static void main(String[] args) {
5.     //Declaring String variable
6.     String levelString="Expert";
7.     int level=0;
8.     //Using String in Switch expression
9.     switch(levelString){
10.    //Using String Literal in Switch case
11.    case "Beginner": level=1;
12.    break;
13.    case "Intermediate": level=2;
14.    break;
15.    case "Expert": level=3;
16.    break;
17.    default: level=0;
18.    break;
19.    }
20.    System.out.println("Your Level is: "+level);
21. }
22. }
3
```

Java Nested Switch Statement

We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

Example:

NestedSwitchExample.java

```
1. //Java Program to demonstrate the use of Java Nested Switch
2. public class NestedSwitchExample {
3.     public static void main(String args[])
4.     {
5.     //C - CSE, E - ECE, M - Mechanical
6.       char branch = 'C';
7.       int collegeYear = 4;
8.       switch( collegeYear )
9.       {
10.          case 1:
11.             System.out.println("English, Maths, Science");
12.             break;
13.          case 2:
14.             switch( branch )
15.             {
16.                case 'C':
17.                   System.out.println("Operating System, Java, Data Structure");
18.                   break;
19.                case 'E':
20.                   System.out.println("Micro processors, Logic switching theory");
21.                   break;
22.                case 'M':
23.                   System.out.println("Drawing, Manufacturing Machines");
24.                   break;
25.             }
26.             break;
27.          case 3:
28.             switch( branch )
29.             {
30.                case 'C':
31.                   System.out.println("Computer Organization, MultiMedia");
32.                   break;
33.                case 'E':
```

```
                System.out.println("Fundamentals of Logic Design, Microelectronics");
                break;
            case 'M':
                System.out.println("Internal Combustion Engines, Mechanical Vibration");
                break;
        }
        break;
      case 4:
        switch( branch )
        {
            case 'C':
                System.out.println("Data Communication and Networks, MultiMedia");
                break;
            case 'E':
                System.out.println("Embedded System, Image Processing");
                break;
            case 'M':
                System.out.println("Production Technology, Thermal Engineering");
                break;
        }
        break;
    }
  }
}
```

ion and Networks, MultiMedia**Java Enum in Switch Statement**

Java allows us to use enum in switch statement. Java enum is a class that represent the group of constants. (immutable such as final variables). We use the keyword enum and put the constants in curly braces separated by comma.

Example:

JavaSwitchEnumExample.java

```
1. //Java Program to demonstrate the use of Enum
2. //in switch statement
3. public class JavaSwitchEnumExample {
```

```
4.      public enum Day {  Sun, Mon, Tue, Wed, Thu, Fri, Sat  }
5.      public static void main(String args[])
6.      {
7.       Day[] DayNow = Day.values();
8.        for (Day Now : DayNow)
9.        {
10.           switch (Now)
11.           {
12.              case Sun:
13.                 System.out.println("Sunday");
14.                 break;
15.              case Mon:
16.                 System.out.println("Monday");
17.                 break;
18.              case Tue:
19.                 System.out.println("Tuesday");
20.                 break;
21.              case Wed:
22.                 System.out.println("Wednesday");
23.                 break;
24.              case Thu:
25.                 System.out.println("Thursday");
26.                 break;
27.              case Fri:
28.                 System.out.println("Friday");
29.                 break;
30.              case Sat:
31.                 System.out.println("Saturday");
32.                 break;
33.           }
34.        }
35.     }
36. }
```

Output:

```
Sunday
Monday
Twesday
Wednesday
Thursday
Friday
Saturday
```

Java Wrapper in Switch Statement

Java allows us to use four wrapper classes: Byte, Short, Integer and Long in switch statement.

Example:

WrapperInSwitchCaseExample.java

```
1. //Java Program to demonstrate the use of Wrapper class
2. //in switch statement
3. public class WrapperInSwitchCaseExample {
4.     public static void main(String args[])
5.     {
6.         Integer age = 18;
7.         switch (age)
8.         {
9.             case (16):
10.                System.out.println("You are under 18.");
11.                break;
12.            case (18):
13.                System.out.println("You are eligible for vote.");
14.                break;
15.            case (65):
16.                System.out.println("You are senior citizen.");
17.                break;
18.            default:
19.                System.out.println("Please give the valid age.");
20.                break;
21.        }
22.    }
```

23. }

Test it Now

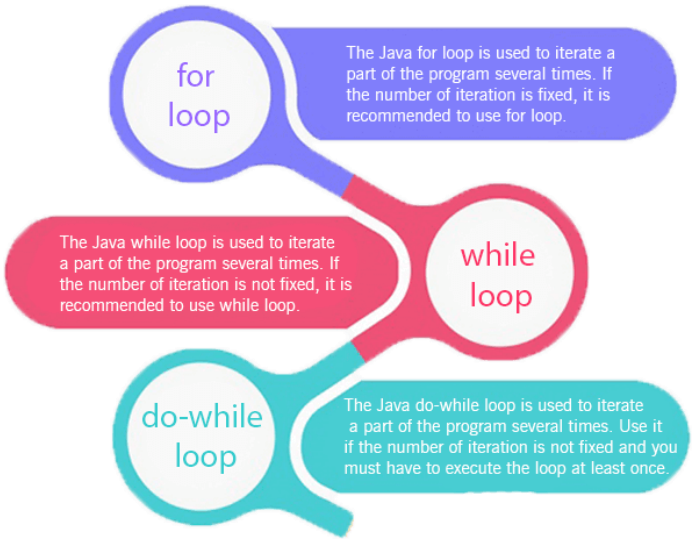
Output:

```
You are eligible for vote.
```

Loops in Java

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is **fixed**, it is recommended to use for loop.

There are three types of for loops in Java.



- Simple for Loop
- For-each or Enhanced for Loop
- Labeled for Loop

Java Simple for Loop

A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

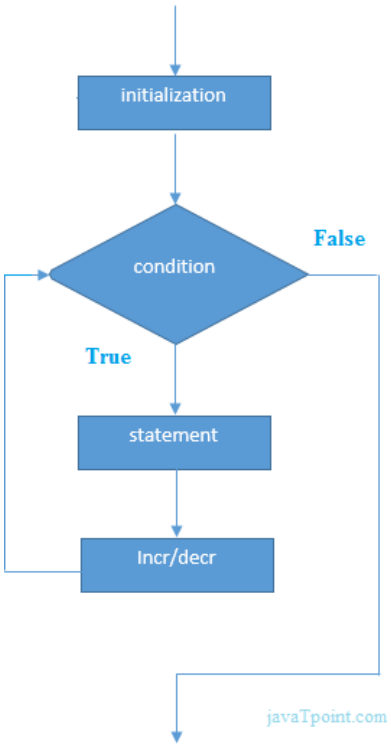
1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.

2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.
4. **Statement**: The statement of the loop is executed each time until the second condition is false.

Syntax:

```
1. for(initialization; condition; increment/decrement){
2. //statement or code to be executed
3. }
```

Flowchart:



Example:

ForExample.java

```
1. //Java Program to demonstrate the example of for loop
2. //which prints table of 1
3. public class ForExample {
4. public static void main(String[] args) {
5.     //Code of Java for loop
6.     for(int i=1;i<=10;i++){
7.         System.out.println(i);
8.     }
9. }
10. }
```

Test it Now

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Java Nested for Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

Example:

NestedForExample.java

```
1. public class NestedForExample {
2. public static void main(String[] args) {
3. //loop of i
4. for(int i=1;i<=3;i++){
5. //loop of j
6. for(int j=1;j<=3;j++){
```

```
7.          System.out.println(i+" "+j);
8.   }//end of i
9.   }//end of j
10. }
11. }
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Pyramid Example 1:

PyramidExample.java

```
1.  public class PyramidExample {
2.  public static void main(String[] args) {
3.  for(int i=1;i<=5;i++){
4.  for(int j=1;j<=i;j++){
5.          System.out.print("* ");
6.  }
7.  System.out.println();//new line
8.  }
9.  }
10. }
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Java for-each Loop

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on the basis of elements and not the index. It returns element one by one in the defined variable.

Syntax:

```
1. for(data_type variable : array_name){
2. //code to be executed
3. }
```

Example:

ForEachExample.java

Java Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop.

Note: The break and continue keywords breaks or continues the innermost for loop respectively.

Syntax:

```
1. labelname:
2. for(initialization; condition; increment/decrement){
3. //code to be executed
4. }
```

Example:

```
3 3
```

Java Infinitive for Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

Syntax:

```
1. for(;;){
```

```
2. //code to be executed
3. }
```

Example:

ForExample.java

```
1. //Java program to demonstrate the use of infinite for loop
2. //which prints an statement
3. public class ForExample {
4. public static void main(String[] args) {
5.     //Using no condition in for loop
6.     for(;;){
7.         System.out.println("infinitive loop");
8.     }
9. }
10. }
```

Output:

```
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
ctrl+c
```

Now, you need to press ctrl+c to exit from the program.

Java for Loop vs while Loop vs do-while Loop

Comparison	for loop	while loop	do-while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon

		boolean condition.	the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

Syntax	for(init;condition;incr/decr){
	// code to be executed
	}
		while(condition){
		//code to be executed
		}
			do{
			//code to be executed
			}while(condition);

Example	//for loop
	for(int i=1;i<=10;i++){
	System.out.println(i);
	}
		//while loop
		int i=1;
		while(i<=10){
		System.out.println(i);
		i++;
		}
			//do-while loop
			int i=1;
			do{
			System.out.println(i);
			i++;
			}while(i<=10);

Syntax for infinitive loop	for(;;){
	//code to be executed
	}
		while(true){
		//code to be executed
		}
			do{
			//code to be executed
			}while(true);

Java While Loop

The Java *while loop* is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while loop.

Syntax:

1. **while** (condition){
2. //code to be executed
3. I ncrement / decrement statement

4. }

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. When the condition becomes false, we exit the while loop.

Example:

i <=100

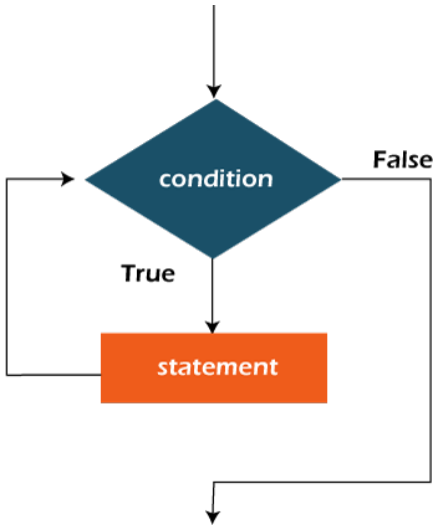
2. Update expression: Every time the loop body is executed, this expression increments or decrements loop variable.

Example:

i++;

Flowchart of Java While Loop

Here, the important thing about while loop is that, sometimes it may not even execute. If the condition to be tested results into false, the loop body is skipped and first statement after the while loop will be executed.



Example:

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

WhileExample.java

```
1. public class WhileExample {
2. public static void main(String[] args) {
3.     int i=1;
4.     while(i<=10){
5.         System.out.println(i);
6.     i++;
7.     }
8. }
9. }
```

Test it Now

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Java Infinitive While Loop

If you pass **true** in the while loop, it will be infinitive while loop.

Syntax:

```
1. while(true){
2. //code to be executed
3. }
```

Example:

WhileExample2.java

```
1. public class WhileExample2 {
2. public static void main(String[] args) {
3.  // setting the infinite while loop by passing true to the condition
4.     while(true){
5.         System.out.println("infinitive while loop");
6.     }
7. }
8. }
```

Output:

```
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
ctrl+c
```

In the above code, we need to enter Ctrl + C command to terminate the infinite loop.

Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
1. do{
2. //code to be executed / loop body
3. //update statement
4. }while (condition);
```

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. As soon as the condition becomes false, loop breaks automatically.

Example:

i <=100

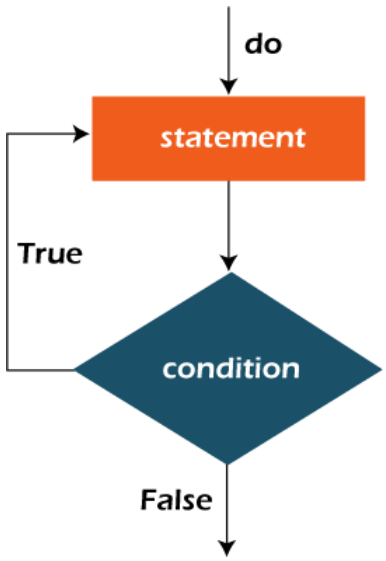
2. Update expression: Every time the loop body is executed, the this expression increments or decrements loop variable.

Example:

i++;

Note: The do block is executed at least once, even if the condition is false.

Flowchart of do-while loop:



Example:

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

DoWhileExample.java

```
1. public class DoWhileExample {
```

```
2. public static void main(String[] args) {
3.     int i=1;
4.     do{
5.         System.out.println(i);
6.     i++;
7.     }while(i<=10);
8. }
9. }
```

Test it Now

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Java Infinitive do-while Loop

If you pass **true** in the do-while loop, it will be infinitive do-while loop.

Syntax:

```
1. do{
2. //code to be executed
3. }while(true);
```

Example:

DoWhileExample2.java

```
1. public class DoWhileExample2 {
2. public static void main(String[] args) {
3.     do{
4.         System.out.println("infinitive do while loop");
```

```
5.      }while(true);
6.  }
7.  }
```

Output:

```
infinitive do while loop
infinitive do while loop
infinitive do while loop
ctrl+c
```

In the above code, we need to enter Ctrl + C command to terminate the infinite loop.

Java Break Statement

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

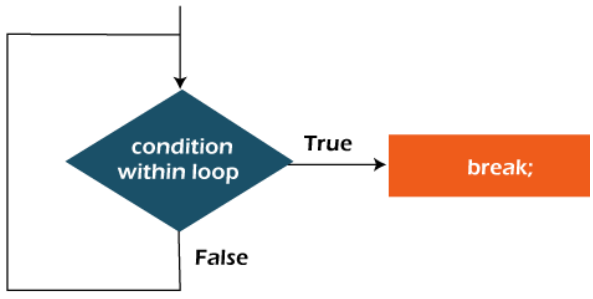
The Java *break* statement is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
1. jump-statement;
2. break;
```

Flowchart of Break Statement



Flowchart of break statement

Java Break Statement with Loop

Example:

BreakExample.java

```
1. //Java Program to demonstrate the use of break statement
2. //inside the for loop.
3. public class BreakExample {
4. public static void main(String[] args) {
5.     //using for loop
6.     for(int i=1;i<=10;i++){
7.         if(i==5){
8.             //breaking the loop
9.             break;
10.        }
11.        System.out.println(i);
12.    }
13. }
14. }
```

Output:

```
1
2
3
4
```

Java Break Statement with Inner Loop

It breaks inner loop only if you use break statement inside the inner loop.

Example:

BreakExample2.java

```
1. //Java Program to illustrate the use of break statement
2. //inside an inner loop
3. public class BreakExample2 {
4. public static void main(String[] args) {
5.             //outer loop
```

```
6.          for(int i=1;i<=3;i++){
7.              //inner loop
8.              for(int j=1;j<=3;j++){
9.                 if(i==2&&j==2){
10.                    //using break statement inside the inner loop
11.                    break;
12.                }
13.                System.out.println(i+" "+j);
14.             }
15.         }
16. }
17. }
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

Java Break Statement with Labeled For Loop

We can use break statement with a label. The feature is introduced since JDK 1.5. So, we can break any loop in Java now whether it is outer or inner loop.

Example:

BreakExample3.java

```
1. //Java Program to illustrate the use of continue statement
2. //with label inside an inner loop to break outer loop
3. public class BreakExample3 {
4. public static void main(String[] args) {
5.          aa:
6.          for(int i=1;i<=3;i++){
7.              bb:
```

```
8.                 for(int j=1;j<=3;j++){
9.                    if(i==2&&j==2){
10.                      //using break statement with label
11.                      break aa;
12.                   }
13.                   System.out.println(i+" "+j);
14.                }
15.          }
16. }
17. }
```

Output:

```
1 1
1 2
1 3
2 1
```

Java Break Statement in while loop

Example:

BreakWhileExample.java

```
1.  //Java Program to demonstrate the use of break statement
2.  //inside the while loop.
3.  public class BreakWhileExample {
4.  public static void main(String[] args) {
5.     //while loop
6.     int i=1;
7.     while(i<=10){
8.        if(i==5){
9.           //using break statement
10.          i++;
11.          break;//it will break the loop
12.       }
13.       System.out.println(i);
14.       i++;
```

```
    }
}
}
```

Output:

```
1
2
3
4
```

Java Break Statement in do-while loop

Example:

BreakDoWhileExample.java

```
//Java Program to demonstrate the use of break statement
//inside the Java do-while loop.
public class BreakDoWhileExample {
public static void main(String[] args) {
    //declaring variable
    int i=1;
    //do-while loop
    do{
        if(i==5){
            //using break statement
            i++;
            break;//it will break the loop
        }
        System.out.println(i);
        i++;
    }while(i<=10);
}
}
```

Output:

```
1
```

```
2
3
4
```

Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
1. jump-statement;
2. continue;
```

Java Continue Statement Example

ContinueExample.java

```
1. //Java Program to demonstrate the use of continue statement
2. //inside the for loop.
3. public class ContinueExample {
4. public static void main(String[] args) {
5.     //for loop
6.     for(int i=1;i<=10;i++){
7.         if(i==5){
8.             //using continue statement
9.             continue;//it will skip the rest statement
10.        }
11.        System.out.println(i);
12.    }
13. }
14. }
```

Output:

```
1
2
3
4
6
7
8
9
10
```

As you can see in the above output, 5 is not printed on the console. It is because the loop is continued when it reaches to 5.

Java Continue Statement with Inner Loop

It continues inner loop only if you use the continue statement inside the inner loop.

ContinueExample2.java

```
1. //Java Program to illustrate the use of continue statement
2. //inside an inner loop
3. public class ContinueExample2 {
4. public static void main(String[] args) {
5.         //outer loop
6.         for(int i=1;i<=3;i++){
7.             //inner loop
8.             for(int j=1;j<=3;j++){
9.                 if(i==2&&j==2){
10.                    //using continue statement inside inner loop
11.                    continue;
12.                }
13.                System.out.println(i+" "+j);
14.            }
15.        }
16. }
17. }
```

Output:

```
1 1
```

```
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

Java Continue Statement with Labelled For Loop

We can use continue statement with a label. This feature is introduced since JDK 1.5. So, we can continue any loop in Java now whether it is outer loop or inner.

Example:

ContinueExample3.java

```
1. //Java Program to illustrate the use of continue statement
2. //with label inside an inner loop to continue outer loop
3. public class ContinueExample3 {
4. public static void main(String[] args) {
5.         aa:
6.         for(int i=1;i<=3;i++){
7.             bb:
8.             for(int j=1;j<=3;j++){
9.                 if(i==2&&j==2){
10.                    //using continue statement with label
11.                    continue aa;
12.                }
13.                System.out.println(i+" "+j);
14.            }
15.        }
16. }
17. }
```

Output:

```
1 1
1 2
1 3
```

```
2 1
3 1
3 2
3 3
```

Java Continue Statement in while loop

ContinueWhileExample.java

```
1. //Java Program to demonstrate the use of continue statement
2. //inside the while loop.
3. public class ContinueWhileExample {
4. public static void main(String[] args) {
5.     //while loop
6.     int i=1;
7.     while(i<=10){
8.         if(i==5){
9.             //using continue statement
10.            i++;
11.            continue;//it will skip the rest statement
12.        }
13.        System.out.println(i);
14.        i++;
15.    }
16. }
17. }
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Java Continue Statement in do-while Loop

ContinueDoWhileExample.java

```
1. //Java Program to demonstrate the use of continue statement
2. //inside the Java do-while loop.
3. public class ContinueDoWhileExample {
4. public static void main(String[] args) {
5.     //declaring variable
6.     int i=1;
7.     //do-while loop
8.     do{
9.         if(i==5){
10.                 //using continue statement
11.                  i++;
12.             continue;//it will skip the rest statement
13.         }
14.         System.out.println(i);
15.         i++;
16.     }while(i<=10);
17. }
18. }
```

Test it Now

Output:

```
1
2
3
4
6
7
8
9
10
```

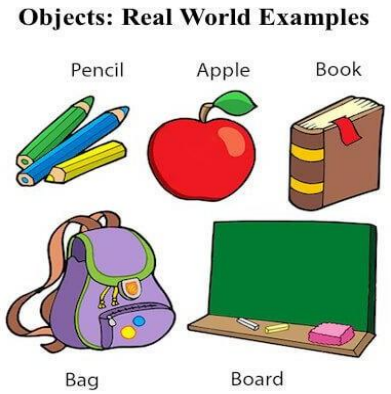
Objects and Classes in Java

1. Object in Java
2. Class in Java
3. Instance Variable in Java
4. Method in Java
5. Example of Object and class that maintains the records of student
6. Anonymous Object

In this page, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

What is an object in Java

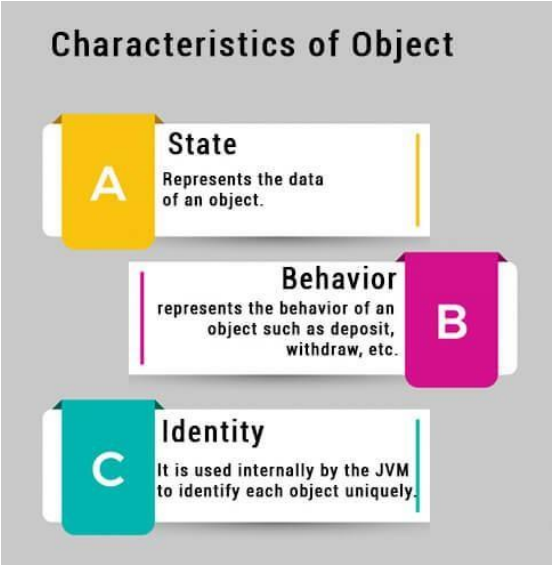


An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.

- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.



For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

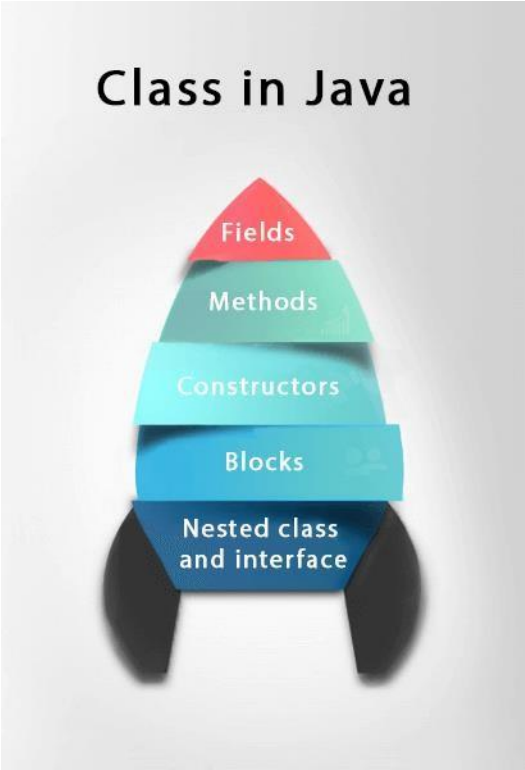
- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**



Syntax to declare a class:

```
1. class <class_name>{
2.     field;
3.     method;
4. }
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

---

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
1. //Java Program to illustrate how to define a class and fields
2. //Defining a Student class.
3. class Student{
4.  //defining fields
5.  int id;//field or data member or instance variable
6.  String name;
7.  //creating main method inside the Student class
```

```
8.   public static void main(String args[]){
9.    //Creating an object or instance
10.   Student s1=new Student();//creating an object of Student
11.   //Printing values of the object
12.   System.out.println(s1.id);//accessing member through reference variable
13.   System.out.println(s1.name);
14.  }
15. }
```

Output:

```
0
null
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```
1.  //Java Program to demonstrate having the main method in
2.  //another class
3.  //Creating Student class.
4.  class Student{
5.   int id;
6.   String name;
7.  }
8.  //Creating another class TestStudent1 which contains the main method
9.  class TestStudent1{
10.  public static void main(String args[]){
11.   Student s1=new Student();
12.   System.out.println(s1.id);
13.   System.out.println(s1.name);
```

```
14.  }
15. }
```

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
1.  class Student{
2.   int id;
3.   String name;
4.  }
5.  class TestStudent2{
6.   public static void main(String args[]){
7.    Student s1=new Student();
8.    s1.id=101;
9.    s1.name="Sonoo";
10.   System.out.println(s1.id+" "+s1.name);//printing members with a white space
11.  }
12. }
```

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent3{
6.  public static void main(String args[]){
7.   //Creating objects
8.   Student s1=new Student();
9.   Student s2=new Student();
10.  //Initializing objects
11.  s1.id=101;
12.  s1.name="Sonoo";
13.  s2.id=102;
14.  s2.name="Amit";
15.  //Printing data
16.  System.out.println(s1.id+"  "+s1.name);
17.  System.out.println(s2.id+"  "+s2.name);
18. }
19. }
```

Test it Now

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

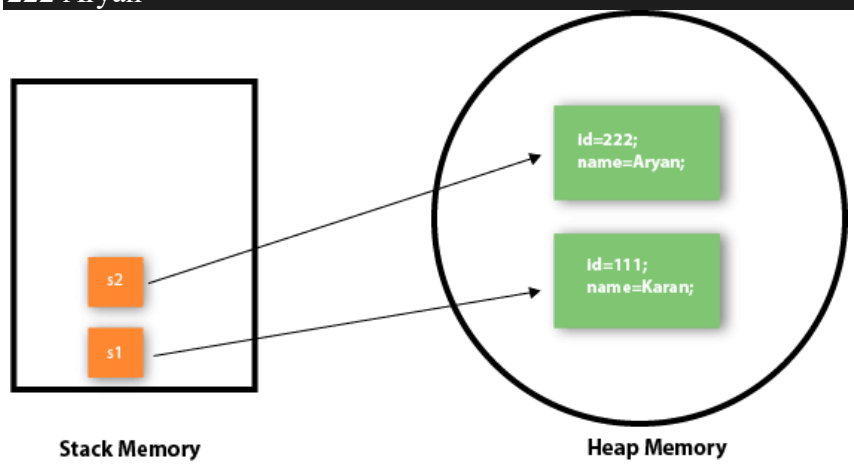
File: TestStudent4.java

```
1. class Student{
```

```
2.   int rollno;
3.   String name;
4.   void insertRecord(int r, String n){
5.    rollno=r;
6.    name=n;
7.   }
8.   void displayInformation(){System.out.println(rollno+" "+name);}
9.  }
10. class TestStudent4{
11.  public static void main(String args[]){
12.   Student s1=new Student();
13.   Student s2=new Student();
14.   s1.insertRecord(111,"Karan");
15.   s2.insertRecord(222,"Aryan");
16.   s1.displayInformation();
17.   s2.displayInformation();
18.  }
19. }
```

Output:

```
111 Karan
222 Aryan
```



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

---

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
1. class Employee{
2.     int id;
3.     String name;
4.     float salary;
5.     void insert(int i, String n, float s) {
6.         id=i;
7.         name=n;
8.         salary=s;
9.     }
10.    void display(){System.out.println(id+" "+name+" "+salary);}
11. }
12. public class TestEmployee {
13. public static void main(String[] args) {
14.    Employee e1=new Employee();
15.    Employee e2=new Employee();
16.    Employee e3=new Employee();
17.    e1.insert(101,"ajeet",45000);
18.    e2.insert(102,"irfan",25000);
19.    e3.insert(103,"nakul",55000);
20.    e1.display();
21.    e2.display();
22.    e3.display();
23. }
```

```
24. }
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```
1.  class Rectangle{
2.   int length;
3.   int width;
4.   void insert(int l, int w){
5.    length=l;
6.    width=w;
7.   }
8.   void calculateArea(){System.out.println(length*width);}
9.  }
10. class TestRectangle1{
11.  public static void main(String args[]){
12.   Rectangle r1=new Rectangle();
13.   Rectangle r2=new Rectangle();
14.   r1.insert(11,5);
15.   r2.insert(3,15);
16.   r1.calculateArea();
17.   r2.calculateArea();
18. }
19. }
```

Output:

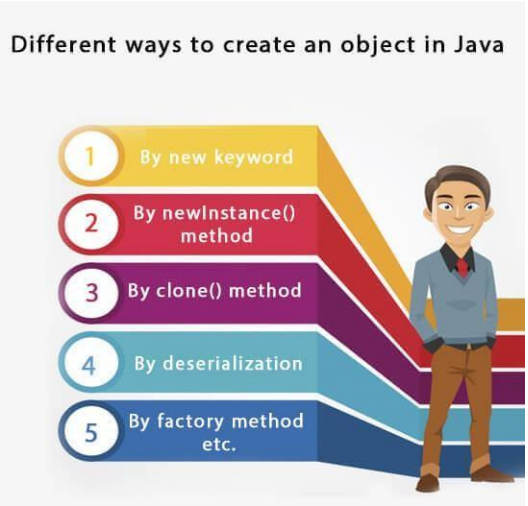
```
55
45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.



Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
1. new Calculation();//anonymous object
```

Calling method through a reference:

```
1. Calculation c=new Calculation();
2. c.fact(5);
```

Calling method through an anonymous object

```
1. new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
1. class Calculation{
2.  void fact(int  n){
3.   int fact=1;
4.   for(int i=1;i<=n;i++){
5.    fact=fact*i;
6.   }
7.  System.out.println("factorial is "+fact);
8. }
9. public static void main(String args[]){
10.  new Calculation().fact(5);//calling method with anonymous object
11. }
12. }
```

Output:

```
Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
1. int a=10, b=20;
```

Initialization of refernce variables:

```
1. Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Let's see the example:

```
1. //Java Program to illustrate the use of Rectangle class which
2. //has length and width data members
3. class Rectangle{
4.  int length;
```

```
5.    int width;
6.    void insert(int l,int w){
7.     length=l;
8.     width=w;
9.    }
10.   void calculateArea(){System.out.println(length*width);}
11. }
12. class TestRectangle2{
13.   public static void main(String args[]){
14.    Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
15.    r1.insert(11,5);
16.    r2.insert(3,15);
17.    r1.calculateArea();
18.    r2.calculateArea();
19. }
20. }
```

Output:

```
55
45
```

Real World Example: Account

File: TestAccount.java

```
1.  //Java Program to demonstrate the working of a banking-system
2.  //where we deposit and withdraw amount from our account.
3.  //Creating an Account class which has deposit() and withdraw() methods
4.  class Account{
5.  int acc_no;
6.  String name;
7.  float amount;
8.  //Method to initialize object
9.  void insert(int a,String n,float amt){
10. acc_no=a;
11. name=n;
```

```
12. amount=amt;
13. }
14. //deposit method
15. void deposit(float amt){
16. amount=amount+amt;
17. System.out.println(amt+" deposited");
18. }
19. //withdraw method
20. void withdraw(float amt){
21. if(amount<amt){
22. System.out.println("Insufficient Balance");
23. }else{
24. amount=amount-amt;
25. System.out.println(amt+" withdrawn");
26. }
27. }
28. //method to check the balance of the account
29. void checkBalance(){System.out.println("Balance is: "+amount);}
30. //method to display the values of an object
31. void display(){System.out.println(acc_no+" "+name+" "+amount);}
32. }
33. //Creating a test class to deposit and withdraw amount
34. class TestAccount{
35. public static void main(String[] args){
36. Account a1=new Account();
37. a1.insert(832345,"Ankit",1000);
38. a1.display();
39. a1.checkBalance();
40. a1.deposit(40000);
41. a1.checkBalance();
42. a1.withdraw(15000);
43. a1.checkBalance();
44. }}
```

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Constructors in Java

1. Types of constructors

1. Default Constructor
2. Parameterized Constructor

2. Constructor Overloading
3. Does constructor return any value?
4. Copying the values of one object into another
5. Does constructor perform other tasks instead of the initialization

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.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

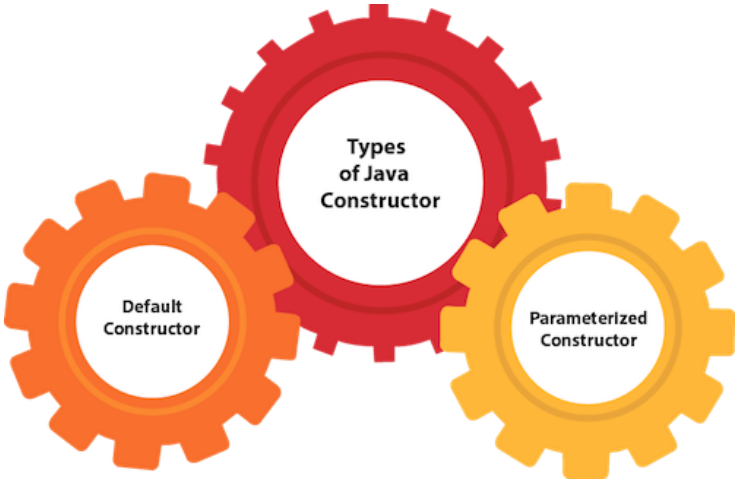
1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

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



Java Default Constructor

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

Syntax of default constructor:

1. <class_name>(){}

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
1. //Java Program to create and call a default constructor
2. class Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
```

```
5. //main method
6. public static void main(String args[]){
7. //calling a default constructor
8. Bike1 b=new Bike1();
9. }
10. }
```

Test it Now

Output:

```
Bike is created
```

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

Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

```
1. //Let us see another example of default constructor
2. //which displays the default values
3. class Student3{
4. int id;
5. String name;
6. //method to display the value of id and name
7. void display(){System.out.println(id+" "+name);}
8.
9. public static void main(String args[]){
```

```
10. //creating objects
11. Student3 s1=new Student3();
12. Student3 s2=new Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }
```

Output:

```
0 null
0 null
```

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

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

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

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.

```
1. //Java Program to demonstrate the use of the parameterized constructor.
2. class Student4{
3.     int id;
4.     String name;
5.     //creating a parameterized constructor
6.     Student4(int i,String n){
7.     id = i;
```

```
8.      name = n;
9.      }
10.     //method to display the values
11.     void display(){System.out.println(id+" "+name);}
12.
13.     public static void main(String args[]){
14.     //creating objects and passing values
15.     Student4 s1 = new Student4(111,"Karan");
16.     Student4 s2 = new Student4(222,"Aryan");
17.     //calling method to display the values of object
18.     s1.display();
19.     s2.display();
20.    }
21. }
```

Test it Now

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

```
1. //Java program to overload constructors
2. class Student5{
3.     int id;
4.     String name;
5.     int age;
```

```
6.      //creating two arg constructor
7.      Student5(int i,String n){
8.      id = i;
9.      name = n;
10.     }
11.     //creating three arg constructor
12.     Student5(int i,String n,int a){
13.     id = i;
14.     name = n;
15.     age=a;
16.     }
17.     void display(){System.out.println(id+" "+name+" "+age);}
18.
19.     public static void main(String args[]){
20.     Student5 s1 = new Student5(111,"Karan");
21.     Student5 s2 = new Student5(222,"Aryan",25);
22.     s1.display();
23.     s2.display();
24.    }
25. }
```

Test it Now

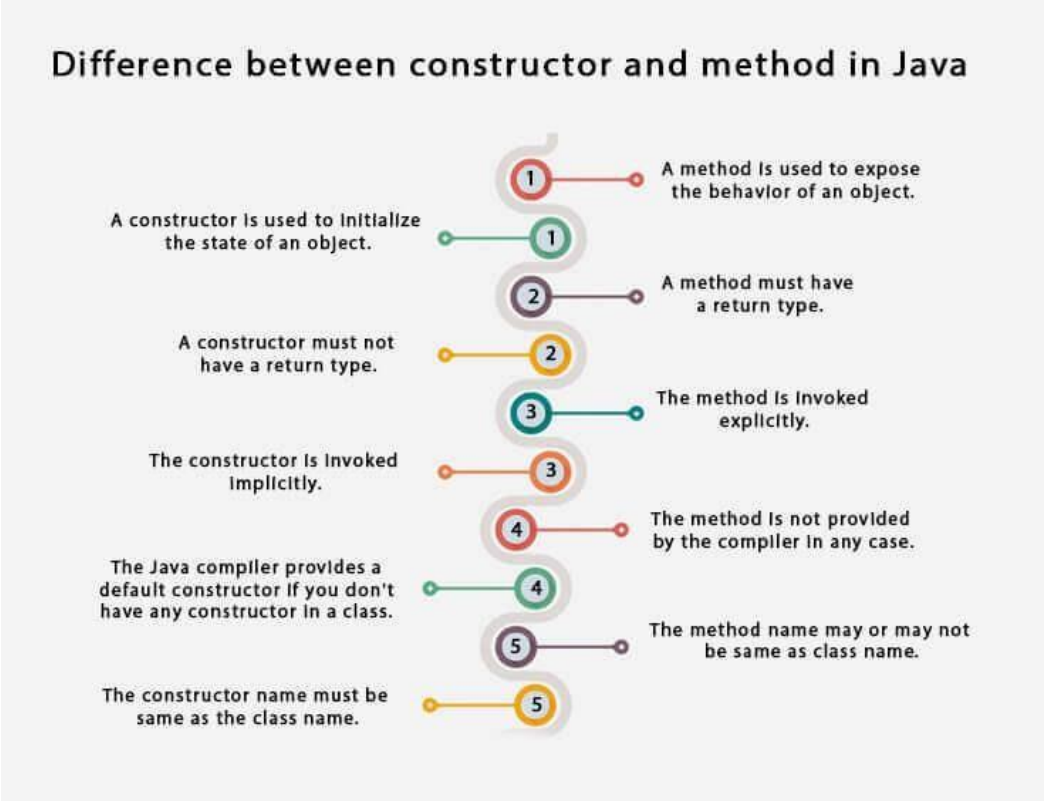
Output:

```
111 Karan 0
222 Aryan 25
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.



Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

```
1. //Java program to initialize the values from one object to another object.
2. class Student6{
3.     int id;
4.     String name;
5.     //constructor to initialize integer and string
6.     Student6(int i,String n){
7.     id = i;
8.     name = n;
9.     }
10.    //constructor to initialize another object
11.    Student6(Student6 s){
12.    id = s.id;
13.    name =s.name;
14.    }
15.    void display(){System.out.println(id+" "+name);}
16.
17.    public static void main(String args[]){
18.    Student6 s1 = new Student6(111,"Karan");
19.    Student6 s2 = new Student6(s1);
20.    s1.display();
21.    s2.display();
22.  }
```

```
23. }
```

Output:

```
111 Karan
111 Karan
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1.  class Student7{
2.      int id;
3.      String name;
4.      Student7(int i,String n){
5.      id = i;
6.      name = n;
7.      }
8.      Student7(){}
9.      void display(){System.out.println(id+" "+name);}
10.
11.     public static void main(String args[]){
12.     Student7 s1 = new Student7(111,"Karan");
13.     Student7 s2 = new Student7();
14.     s2.id=s1.id;
15.     s2.name=s1.name;
16.     s1.display();
17.     s2.display();
18.    }
19. }
```

Output:

```
111 Karan
111 Karan
```

Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Is there Constructor class in Java?

Yes.

What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

Java 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.

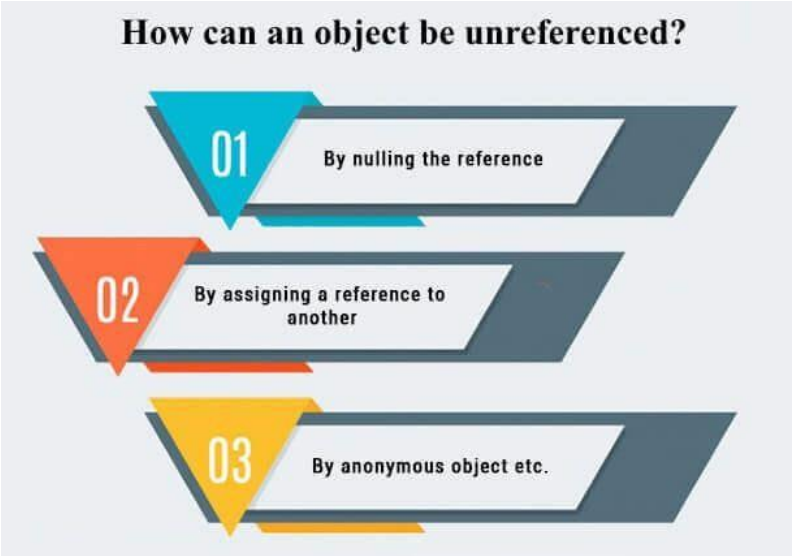
Advantage of Garbage Collection

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

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.



1) By nulling a reference:

```
1. Employee e=new Employee();
2. e=null;
```

2) By assigning a reference to another:

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

3) By anonymous object:

```
1. 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:

```
1. 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.

```
1. public static void gc(){}
```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
1. public class TestGarbage1{
2.  public void finalize(){System.out.println("object is garbage collected");}
3.  public static void main(String args[]){
4.   TestGarbage1 s1=new TestGarbage1();
5.   TestGarbage1 s2=new TestGarbage1();
6.   s1=null;
7.   s2=null;
8.   System.gc();
9.  }
10. }
```

Test it Now

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

Java static keyword

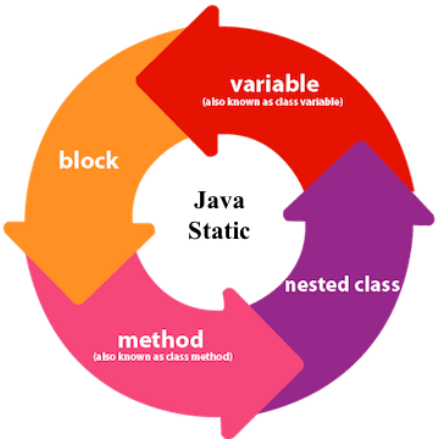
1. Static variable
2. Program of the counter without static variable
3. Program of the counter with static variable

4. Static method
5. Restrictions for the static method
6. Why is the main method static?
7. Static block
8. Can we execute a program without main method?

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Understanding the problem without static variable

```
1. class Student{
2.     int rollno;
3.     String name;
4.     String college="ITS";
5. }
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Java static property is shared to all objects.

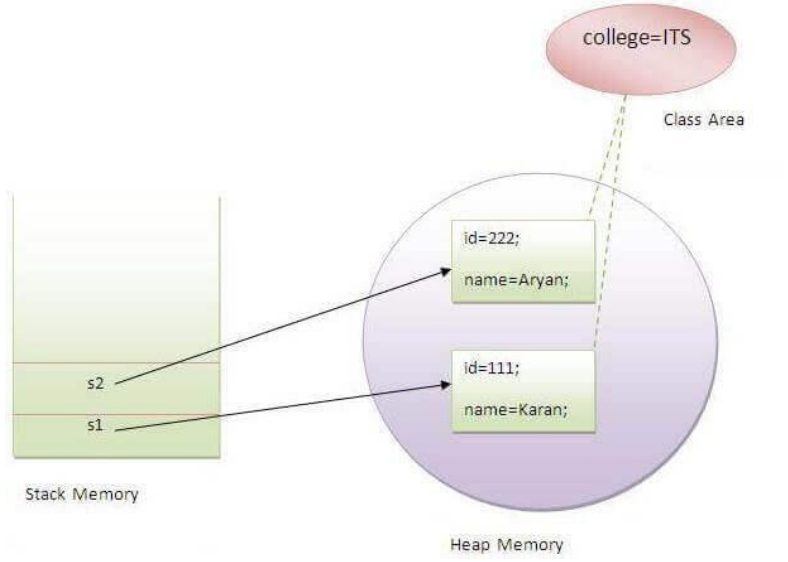
Example of static variable

```
1. //Java Program to demonstrate the use of static variable
2. class Student{
3.   int rollno;//instance variable
4.   String name;
5.   static String college ="ITS";//static variable
6.   //constructor
7.   Student(int r, String n){
8.   rollno = r;
9.   name = n;
10.  }
11.  //method to display the values
12.  void display (){System.out.println(rollno+" "+name+" "+college);}
13. }
14. //Test class to show the values of objects
15. public class TestStaticVariable1{
16.  public static void main(String args[]){
17.  Student s1 = new Student(111,"Karan");
```

```
18.  Student s2 = new Student(222,"Aryan");
19.  //we can change the college of all objects by the single line of code
20.  //Student.college="BBDIT";
21.  s1.display();
22.  s2.display();
23.  }
24. }
```

Output:

```
111 Karan ITS
222 Aryan ITS
```



Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
1. //Java Program to demonstrate the use of an instance variable
2. //which get memory each time when we create an object of the class.
3. class Counter{
4. int count=0;//will get memory each time when the instance is created
5.
```

```
6. Counter(){
7. count++;//incrementing value
8. System.out.println(count);
9. }
10.
11. public static void main(String args[]){
12. //Creating objects
13. Counter c1=new Counter();
14. Counter c2=new Counter();
15. Counter c3=new Counter();
16. }
17. }
```

Test it Now

Output:

```
1
1
1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
1. //Java Program to illustrate the use of static variable which
2. //is shared with all objects.
3. class Counter2{
4. static int count=0;//will get memory only once and retain its value
5.
6. Counter2(){
7. count++;//incrementing the value of static variable
8. System.out.println(count);
9. }
10.
11. public static void main(String args[]){
12. //creating objects
```

```
13. Counter2 c1=new Counter2();
14. Counter2 c2=new Counter2();
15. Counter2 c3=new Counter2();
16. }
17. }
```

Test it Now

Output:

```
1
2
3
```

---

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
1.  //Java Program to demonstrate the use of a static method.
2.  class Student{
3.      int rollno;
4.      String name;
5.      static String college = "ITS";
6.      //static method to change the value of static variable
7.      static void change(){
8.      college = "BBDIT";
9.      }
10.     //constructor to initialize the variable
11.     Student(int r, String n){
12.     rollno = r;
13.     name = n;
14.     }
```

```
15.      //method to display values
16.      void display(){System.out.println(rollno+" "+name+" "+college);}
17. }
18. //Test class to create and display the values of object
19. public class TestStaticMethod{
20.     public static void main(String args[]){
21.     Student.change();//calling change method
22.     //creating objects
23.     Student s1 = new Student(111,"Karan");
24.     Student s2 = new Student(222,"Aryan");
25.     Student s3 = new Student(333,"Sonoo");
26.     //calling display method
27.     s1.display();
28.     s2.display();
29.     s3.display();
30.     }
31. }
```

Test it Now

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

```
1.  //Java Program to get the cube of a given number using the static method
2.
3.   class Calculate{
4.     static int cube(int x){
5.     return x*x*x;
6.     }
7.
8.     public static void main(String args[]){
9.     int result=Calculate.cube(5);
10.    System.out.println(result);
11.    }
```

```
12. }
```
Output:125

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
1. class A{
2.  int a=40;//non static
3.
4.  public static void main(String args[]){
5.   System.out.println(a);
6.  }
7. }
```
Output:Compile Time Error

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

---

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
1. class A2{
2.   static{System.out.println("static block is invoked");}
3.   public static void main(String args[]){
4.    System.out.println("Hello main");
```

```
5.   }
6. }
```

```
Output:static block is invoked
       Hello main
```

Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the main method.

```
1. class A3{
2.   static{
3.   System.out.println("static block is invoked");
4.   System.exit(0);
5.   }
6. }
```

Output:

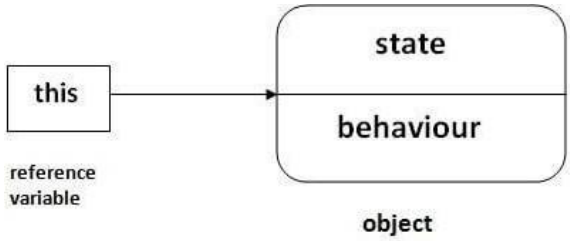
```
static block is invoked
```

Since JDK 1.7 and above, output would be:

```
Error: Main method not found in class A3, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

this keyword in Java

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.

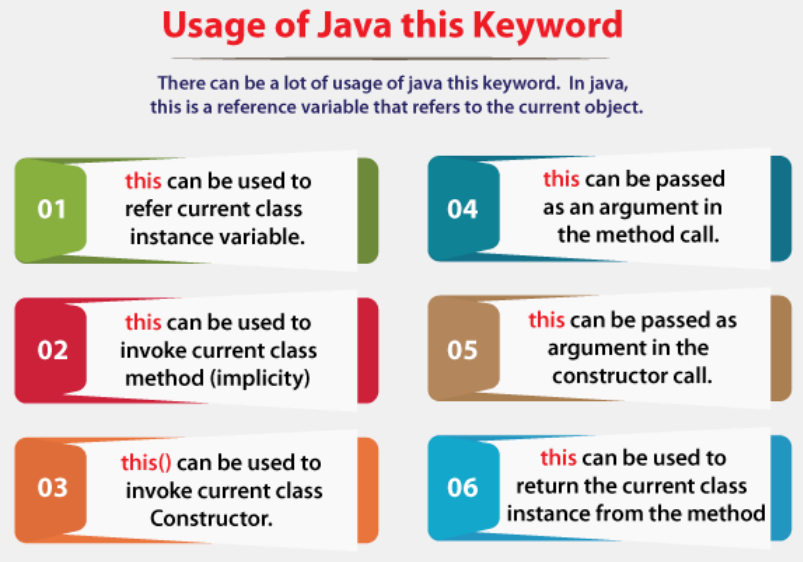


Usage of Java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

Suggestion: If you are beginner to java, lookup only three usages of this keyword.



---

1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
1. class Student{
2. int rollno;
```

```
3. String name;
4. float fee;
5. Student(int rollno,String name,float fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. class TestThis1{
13. public static void main(String args[]){
14. Student s1=new Student(111,"ankit",5000f);
15. Student s2=new Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}
```

Output:

```
0 null 0.0
0 null 0.0
```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
1. class Student{
2. int rollno;
3. String name;
4. float fee;
5. Student(int rollno,String name,float fee){
6. this.rollno=rollno;
7. this.name=name;
8. this.fee=fee;
9. }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
```

```
11. }
12.
13. class TestThis2{
14. public static void main(String args[]){
15. Student s1=new Student(111,"ankit",5000f);
16. Student s2=new Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}
```

Test it Now

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

```
1. class Student{
2. int rollno;
3. String name;
4. float fee;
5. Student(int r,String n,float f){
6. rollno=r;
7. name=n;
8. fee=f;
9. }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12.
13. class TestThis3{
14. public static void main(String args[]){
15. Student s1=new Student(111,"ankit",5000f);
16. Student s2=new Student(112,"sumit",6000f);
```

17. s1.display();
18. s2.display();
19. }}

Output:

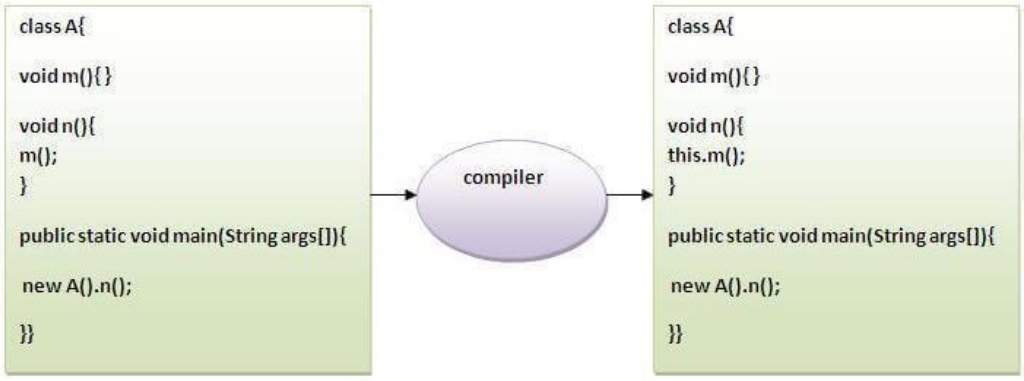
```
111 ankit 5000.0
112 sumit 6000.0
```

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

---

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



1. **class** A{
2. **void** m(){System.out.println("hello m");}
3. **void** n(){
4. System.out.println("hello n");
5. //m();//same as this.m()
6. **this**.m();
7. }
8. }
9. **class** TestThis4{

```
10. public static void main(String args[]){
11. A a=new A();
12. a.n();
13. }}
```

Test it Now

Output:

```
hello n
hello m
```

---

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

```
1. class A{
2. A(){System.out.println("hello a");}
3. A(int x){
4. this();
5. System.out.println(x);
6. }
7. }
8. class TestThis5{
9. public static void main(String args[]){
10. A a=new A(10);
11. }}
```

Output:

```
hello a
10
```

Calling parameterized constructor from default constructor:

```
1. class A{
2. A(){
```

```
3. this(5);
4. System.out.println("hello a");
5. }
6. A(int x){
7. System.out.println(x);
8. }
9. }
10. class TestThis6{
11. public static void main(String args[]){
12. A a=new A();
13. }}
```

Output:

```
5
hello a
```

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
1. class Student{
2. int rollno;
3. String name,course;
4. float fee;
5. Student(int rollno,String name,String course){
6. this.rollno=rollno;
7. this.name=name;
8. this.course=course;
9. }
10. Student(int rollno,String name,String course,float fee){
11. this(rollno,name,course);//reusing constructor
12. this.fee=fee;
13. }
14. void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
```

```
15. }
16. class TestThis7{
17. public static void main(String args[]){
18. Student s1=new Student(111,"ankit","java");
19. Student s2=new Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}
```

Output:

```
111 ankit java 0.0
112 sumit java 6000.0
```

Rule: Call to this() must be the first statement in constructor.

```
1. class Student{
2. int rollno;
3. String name,course;
4. float fee;
5. Student(int rollno,String name,String course){
6. this.rollno=rollno;
7. this.name=name;
8. this.course=course;
9. }
10. Student(int rollno,String name,String course,float fee){
11. this.fee=fee;
12. this(rollno,name,course);//C.T.Error
13. }
14. void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. class TestThis8{
17. public static void main(String args[]){
18. Student s1=new Student(111,"ankit","java");
19. Student s2=new Student(112,"sumit","java",6000f);
20. s1.display();
```

```
21. s2.display();
22. }}
```

Output:

```
Compile Time Error: Call to this must be first statement in constructor
```

---

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
1.  class S2{
2.    void m(S2 obj){
3.    System.out.println("method is invoked");
4.    }
5.    void p(){
6.    m(this);
7.    }
8.    public static void main(String args[]){
9.    S2 s1 = new S2();
10.   s1.p();
11.   }
12. }
```

Output:

```
method is invoked
```

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

---

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
1. class B{
2.   A4 obj;
3.   B(A4 obj){
4.     this.obj=obj;
5.   }
6.   void display(){
7.     System.out.println(obj.data);//using data member of A4 class
8.   }
9. }
10.
11. class A4{
12.   int data=10;
13.   A4(){
14.    B b=new B(this);
15.    b.display();
16.   }
17.   public static void main(String args[]){
18.    A4 a=new A4();
19.   }
20. }
```

Output:10

---

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
1. return_type method_name(){
2. return this;
3. }
```

Example of this keyword that you return as a statement from the method

```
1. class A{
```

```
2. A getA(){
3. return this;
4. }
5. void msg(){System.out.println("Hello java");}
6. }
7. class Test1{
8. public static void main(String args[]){
9. new A().getA().msg();
10. }
11. }
```

Output:

```
Hello java
```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
1. class A5{
2. void m(){
3. System.out.println(this);//prints same reference ID
4. }
5. public static void main(String args[]){
6. A5 obj=new A5();
7. System.out.println(obj);//prints the reference ID
8. obj.m();
9. }
10. }
```

Output:

```
A5@22b3ea59
A5@22b3ea59
```

Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

1. arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. class Testarray{
4. public static void main(String args[]){
5. int a[]=new int[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. for(int i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
1. int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
1. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. class Testarray1{
4. public static void main(String args[]){
5. int a[]={33,3,4,5};//declaration, instantiation and initialization
6. //printing array
7. for(int i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
9. }}
```

Test it Now

Output:

```
33
3
4
5
```

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
1. for(data_type variable:array){
2. //body of the loop
3. }
```

Let us see the example of print the elements of Java array using the for-each loop.

```
1. //Java Program to print the array elements using for-each loop
2. class Testarray1{
```

```
3. public static void main(String args[]){
4. int arr[]={33,3,4,5};
5. //printing array using for-each loop
6. for(int i:arr)
7. System.out.println(i);
8. }}
```

Output:

```
33
3
4
5
```

Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get the minimum number of an array using a method.

```
1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. class Testarray2{
4. //creating a method which receives an array as a parameter
5. static void min(int arr[]){
6. int min=arr[0];
7. for(int i=1;i<arr.length;i++)
8.  if(min>arr[i])
9.   min=arr[i];
10.
11. System.out.println(min);
12. }
13.
14. public static void main(String args[]){
15. int a[]={33,3,4,5};//declaring and initializing an array
16. min(a);//passing array to method
17. }}
```

Output:

```
3
```

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
1. //Java Program to demonstrate the way of passing an anonymous array
2. //to method.
3. public class TestAnonymousArray{
4. //creating a method which receives an array as a parameter
5. static void printArray(int arr[]){
6. for(int i=0;i<arr.length;i++)
7. System.out.println(arr[i]);
8. }
9.
10. public static void main(String args[]){
11. printArray(new int[]{10,22,44,66});//passing anonymous array to method
12. }}
```

Output:

```
10
22
44
66
```

Returning Array from the Method

We can also return an array from the method in Java.

```
1. //Java Program to return an array from the method
2. class TestReturnArray{
3. //creating method which returns an array
4. static int[] get(){
5. return new int[]{10,30,50,90,60};
6. }
```

```
7.
8.  public static void main(String args[]){
9.  //calling method which returns an array
10. int arr[]=get();
11. //printing the values of an array
12. for(int i=0;i<arr.length;i++)
13. System.out.println(arr[i]);
14. }}
```

Output:

```
10
30
50
90
60
```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

```
1.  //Java Program to demonstrate the case of
2.  //ArrayIndexOutOfBoundsException in a Java Array.
3.  public class TestArrayException{
4.  public static void main(String args[]){
5.  int arr[]={50,60,70,80};
6.  for(int i=0;i<=arr.length;i++){
7.  System.out.println(arr[i]);
8.  }
9.  }}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at TestArrayException.main(TestArrayException.java:5)
50
60
70
80
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
1. int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
1. //Java Program to illustrate the use of multidimensional array
2. class Testarray3{
3. public static void main(String args[]){
4. //declaring and initializing 2D array
5. int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6. //printing 2D array
```

```
7.  for(int i=0;i<3;i++){
8.   for(int j=0;j<3;j++){
9.     System.out.print(arr[i][j]+"  ");
10.  }
11.  System.out.println();
12. }
13. }}
```

Output:

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

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
1.  //Java Program to illustrate the jagged array
2.  class TestJaggedArray{
3.     public static void main(String[] args){
4.        //declaring a 2D array with odd columns
5.        int arr[][] = new int[3][];
6.        arr[0] = new int[3];
7.        arr[1] = new int[4];
8.        arr[2] = new int[2];
9.        //initializing a jagged array
10.       int count = 0;
11.       for (int i=0; i<arr.length; i++)
12.          for(int j=0; j<arr[i].length; j++)
13.             arr[i][j] = count++;
14.
15.       //printing the data of a jagged array
16.       for (int i=0; i<arr.length; i++){
17.          for (int j=0; j<arr[i].length; j++){
18.             System.out.print(arr[i][j]+"  ");
```

```
            }
            System.out.println();//new line
        }
    }
}
```

Output:

```
0 1 2
3 4 5 6
7 8
```

What is the class name of Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by getClass().getName() method on the object.

```
//Java Program to get the class name of array in Java
class Testarray4{
public static void main(String args[]){
//declaration and initialization of array
int arr[]={4,4,5};
//getting the class name of Java array
Class c=arr.getClass();
String name=c.getName();
//printing the class name of Java array
System.out.println(name);

}}
```

Output:

```
I
```

Copying a Java Array

We can copy an array to another by the arraycopy() method of System class.

Syntax of arraycopy method

```
1. public static void arraycopy(
2. Object src, int srcPos,Object dest, int destPos, int length
3. )
```

Example of Copying an Array in Java

```
1. //Java Program to copy a source array into a destination array in Java
2. class TestArrayCopyDemo {
3.     public static void main(String[] args) {
4.         //declaring a source array
5.         char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
6.                 'i', 'n', 'a', 't', 'e', 'd' };
7.         //declaring a destination array
8.         char[] copyTo = new char[7];
9.         //copying array using System.arraycopy() method
10.        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
11.        //printing the destination array
12.        System.out.println(String.valueOf(copyTo));
13.    }
14. }
```

Output:

```
caffein
```

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

```
1. //Java Program to clone the array
2. class Testarray1{
3. public static void main(String args[]){
```

```
4. int arr[]={33,3,4,5};
5. System.out.println("Printing original array:");
6. for(int i:arr)
7. System.out.println(i);
8.
9. System.out.println("Printing clone of the array:");
10. int carr[]=arr.clone();
11. for(int i:carr)
12. System.out.println(i);
13.
14. System.out.println("Are both equal?");
15. System.out.println(arr==carr);
16.
17. }}
```

Output:

```
Printing original array:
33
3
4
5
Printing clone of the array:
33
3
4
5
Are both equal?
false
```

Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

```
1. //Java Program to demonstrate the addition of two matrices in Java
2. class Testarray5{
3. public static void main(String args[]){
4. //creating two matrices
5. int a[][]={{1,3,4},{3,4,5}};
```

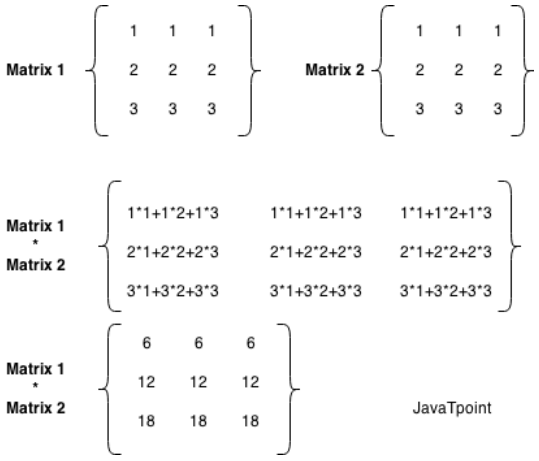
```
6. int b[][]={{1,3,4},{3,4,5}};
7.
8. //creating another matrix to store the sum of two matrices
9. int c[][]=new int[2][3];
10.
11. //adding and printing addition of 2 matrices
12. for(int i=0;i<2;i++){
13. for(int j=0;j<3;j++){
14. c[i][j]=a[i][j]+b[i][j];
15. System.out.print(c[i][j]+" ");
16. }
17. System.out.println();//new line
18. }
19.
20. }}
```

Output:

```
2 6 8
6 8 10
```

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.



Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

```
1. //Java Program to multiply two matrices
2. public class MatrixMultiplicationExample{
3. public static void main(String args[]){
4. //creating two matrices
5. int a[][]={{1,1,1},{2,2,2},{3,3,3}};
6. int b[][]={{1,1,1},{2,2,2},{3,3,3}};
7.
8. //creating another matrix to store the multiplication of two matrices
9. int c[][]=new int[3][3];  //3 rows and 3 columns
10.
11. //multiplying and printing multiplication of 2 matrices
12. for(int i=0;i<3;i++){
13. for(int j=0;j<3;j++){
14. c[i][j]=0;
15. for(int k=0;k<3;k++)
16. {
17. c[i][j]+=a[i][k]*b[k][j];
18. }//end of k loop
19. System.out.print(c[i][j]+" ");  //printing matrix element
20. }//end of j loop
21. System.out.println();//new line
22. }
23. }}
```

Output:

```
6 6 6
12 12 12
18 18 18
```

Java Command Line Arguments

1. Command Line Argument
2. Simple example of command-line argument
3. Example of command-line argument that prints all the values

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pas least one argument from the command prompt.

```
1. class CommandLineExample{
2. public static void main(String args[]){
3. System.out.println("Your first argument is: "+args[0]);
4. }
5. }
```

```
1. compile by > javac CommandLineExample.java
2. run by > java CommandLineExample sonoo
```

```
Output: Your first argument is: sonoo
```

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we h traversed the array using for loop.

```
1. class A{
2. public static void main(String args[]){
3.
4. for(int i=0;i<args.length;i++)
5. System.out.println(args[i]);
6.
7. }
8. }
```

```
1. compile by > javac A.java
2. run by > java A sonoo jaiswal 1 3 abc
```

```
Output: sonoo
        jaiswal
        1
        3
        abc
```

Inheritance in Java

1. Inheritance
2. Types of Inheritance
3. Why multiple inheritance is not possible in Java in case of class?

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

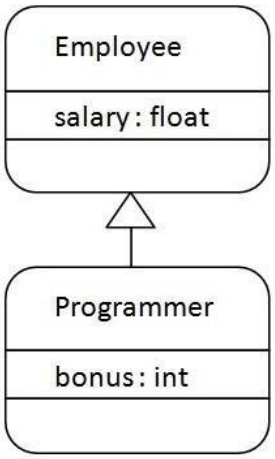
```
1. class Subclass-name extends Superclass-name
2. {
3.   //methods and fields
```

```
4. }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
1. class Employee{
2.  float salary=40000;
3. }
4. class Programmer extends Employee{
5.  int bonus=10000;
6.  public static void main(String args[]){
7.    Programmer p=new Programmer();
8.    System.out.println("Programmer salary is:"+p.salary);
9.    System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }
```

```
Programmer salary is:40000.0
```

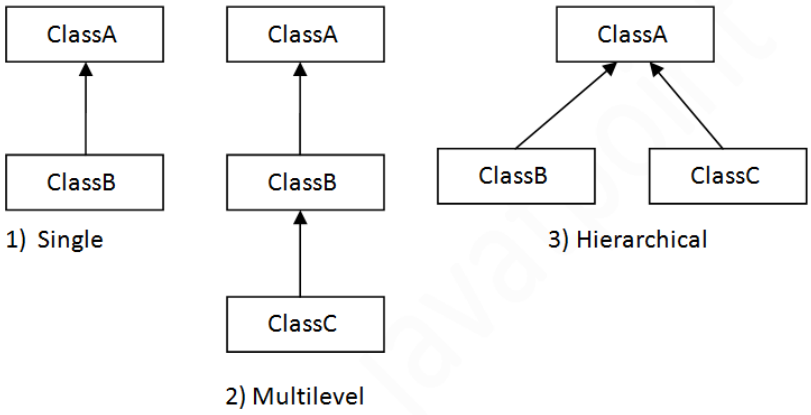
Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

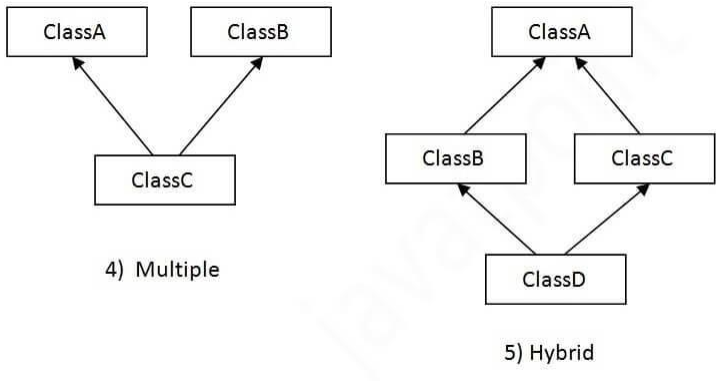
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



.

Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
```

```
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
8. void meow(){System.out.println("meowing...");}
9. }
10. class TestInheritance3{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
```

```
14. c.eat();
15. //c.bark();//C.T.Error
16. }}
```

Output:

```
meowing...
eating...
```

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
1. class A{
2. void msg(){System.out.println("Hello");}
3. }
4. class B{
5. void msg(){System.out.println("Welcome");}
6. }
7. class C extends A,B{//suppose if it were
8.
9.  public static void main(String args[]){
10.   C obj=new C();
11.   obj.msg();//Now which msg() method would be invoked?
12. }
13. }
```

```
Compile Time Error
```

Method Overloading in Java

1. Different ways to overload the method
2. By changing the no. of arguments
3. By changing the datatype
4. Why method overloading is not possible by changing the return type
5. Can we overload the main method
6. method overloading with Type Promotion

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In Java, Method Overloading is not possible by changing the return type of the method only.

---

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
5. class TestOverloading1{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}
```

Output:

```
22
33
```

---

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
1. class Adder{
2. static int add(int a, int b){return a+b;}
3. static double add(double a, double b){return a+b;}
4. }
5. class TestOverloading2{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}
```

Output:

```
22
24.9
```

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static double add(int a,int b){return a+b;}
4. }
5. class TestOverloading3{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));//ambiguity
8. }}
```

Output:

```
Compile Time Error: method add(int,int) is already defined in class Adder
```

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

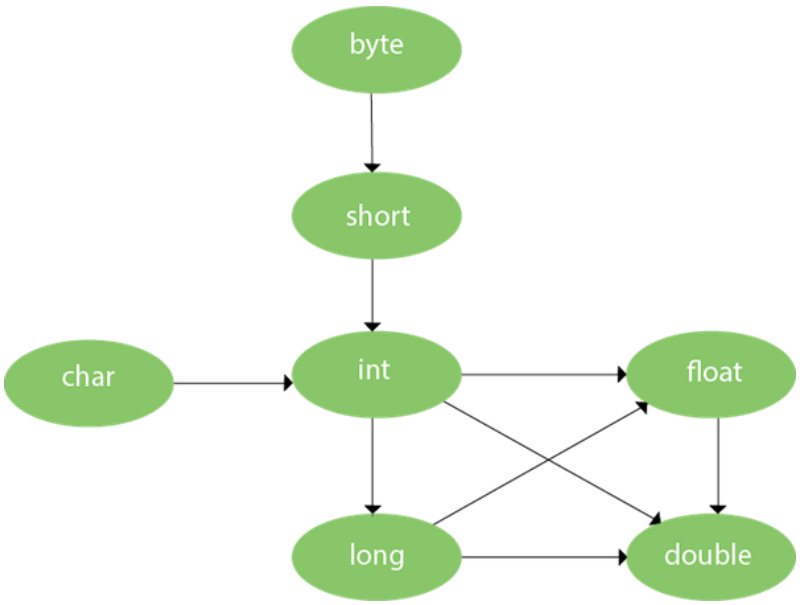
```
1. class TestOverloading4{
2. public static void main(String[] args){System.out.println("main with String[]");}
3. public static void main(String args){System.out.println("main with String");}
4. public static void main(){System.out.println("main without args");}
5. }
```

Output:

```
main with String[]
```

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
1. class OverloadingCalculation1{
2.   void sum(int a,long b){System.out.println(a+b);}
3.   void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.   public static void main(String args[]){
6.   OverloadingCalculation1 obj=new OverloadingCalculation1();
7.   obj.sum(20,20);//now second int literal will be promoted to long
8.   obj.sum(20,20,20);
9.
10.  }
```

```
11. }
```

Test it Now

```
Output:40
       60
```

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
1.  class OverloadingCalculation2{
2.    void sum(int a,int b){System.out.println("int arg method invoked");}
3.    void sum(long a,long b){System.out.println("long arg method invoked");}
4.
5.    public static void main(String args[]){
6.    OverloadingCalculation2 obj=new OverloadingCalculation2();
7.    obj.sum(20,20);//now int arg sum() method gets invoked
8.    }
9.  }
```

```
Output:int arg method invoked
```

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
1.  class OverloadingCalculation3{
2.    void sum(int a,long b){System.out.println("a method invoked");}
3.    void sum(long a,int b){System.out.println("b method invoked");}
4.
5.    public static void main(String args[]){
6.    OverloadingCalculation3 obj=new OverloadingCalculation3();
7.    obj.sum(20,20);//now ambiguity
8.    }
9.  }
```

```
Output:Compile Time Error
```

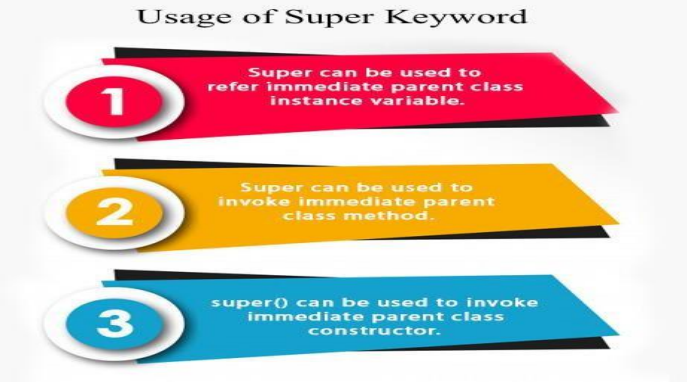
Super Keyword in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.



1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
1. class Animal{
2. String color="white";
3. }
4. class Dog extends Animal{
5. String color="black";
6. void printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(super.color);//prints color of Animal class
9. }
10. }
```

```
11. class TestSuper1{
12. public static void main(String args[]){
13. Dog d=new Dog();
14. d.printColor();
15. }}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10. }
11. }
12. class TestSuper2{
13. public static void main(String args[]){
14. Dog d=new Dog();
15. d.work();
16. }}
```

Output:

```
eating...
barking...
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

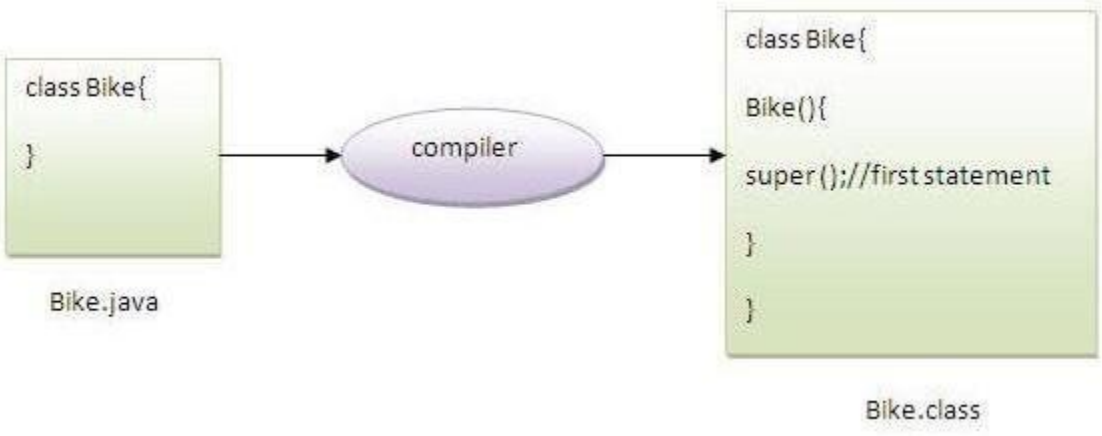
The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. super();
7. System.out.println("dog is created");
8. }
9. }
10. class TestSuper3{
11. public static void main(String args[]){
12. Dog d=new Dog();
13. }}
```

Output:

```
animal is created
dog is created
```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. System.out.println("dog is created");
7. }
8. }
9. class TestSuper4{
10. public static void main(String args[]){
11. Dog d=new Dog();
12. }}
```

Output:

```
animal is created
dog is created
```

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are

using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
1. class Person{
2. int id;
3. String name;
4. Person(int id,String name){
5. this.id=id;
6. this.name=name;
7. }
8. }
9. class Emp extends Person{
10. float salary;
11. Emp(int id,String name,float salary){
12. super(id,name);//reusing parent constructor
13. this.salary=salary;
14. }
15. void display(){System.out.println(id+" "+name+" "+salary);}
16. }
17. class TestSuper5{
18. public static void main(String[] args){
19. Emp e1=new Emp(1,"ankit",45000f);
20. e1.display();
21. }}
```

Output:

```
1 ankit 45000
```

Final Keyword In Java

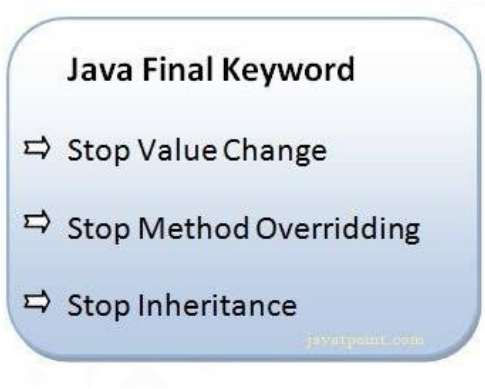
1. Final variable
2. Final method
3. Final class
4. Is final method inherited ?
5. Blank final variable
6. Static blank final variable

7. Final parameter
8. Can you declare a final constructor

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.



1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1. class Bike9{
2.  final int speedlimit=90;//final variable
3.  void run(){
4.   speedlimit=400;
5.  }
```

```
6.   public static void main(String args[]){
7.   Bike9 obj=new  Bike9();
8.   obj.run();
9.   }
10. }//end of class
```

Output:Compile Time Error

---

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
1.  class Bike{
2.    final void run(){System.out.println("running");}
3.  }
4.
5.  class Honda extends Bike{
6.     void run(){System.out.println("running safely with 100kmph");}
7.
8.     public static void main(String args[]){
9.     Honda honda= new Honda();
10.    honda.run();
11.    }
12. }
```

Output:Compile Time Error

---

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
1.  final class Bike{}
2.
3.  class Honda1 extends Bike{
4.    void run(){System.out.println("running safely with 100kmph");}
```

```
5.
6.    public static void main(String args[]){
7.    Honda1 honda= new Honda1();
8.    honda.run();
9.    }
10. }
```

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1. class Bike{
2.   final void run(){System.out.println("running...");}
3. }
4. class Honda2 extends Bike{
5.    public static void main(String args[]){
6.     new Honda2().run();
7.    }
8. }
```

Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
1. class Student{
2. int id;
3. String name;
4. final String PAN_CARD_NUMBER;
```

```
5. ...
6. }
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
1. class Bike10{
2.   final int speedlimit;//blank final variable
3.
4.   Bike10(){
5.   speedlimit=70;
6.   System.out.println(speedlimit);
7.   }
8.
9.   public static void main(String args[]){
10.    new Bike10();
11. }
12. }
```

Output: 70

---

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
1. class A{
2.   static final int data;//static blank final variable
3.   static{ data=50;}
4.   public static void main(String args[]){
5.     System.out.println(A.data);
6.  }
7. }
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
1. class Bike11{
2.   int cube(final int n){
3.    n=n+2;//can't be changed as n is final
4.    n*n*n;
5.   }
6.   public static void main(String args[]){
7.     Bike11 b=new Bike11();
8.     b.cube(5);
9.  }
10. }
```

Output: Compile Time Error

Q) Can we declare a constructor final?

No, because constructor is never inherited.

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

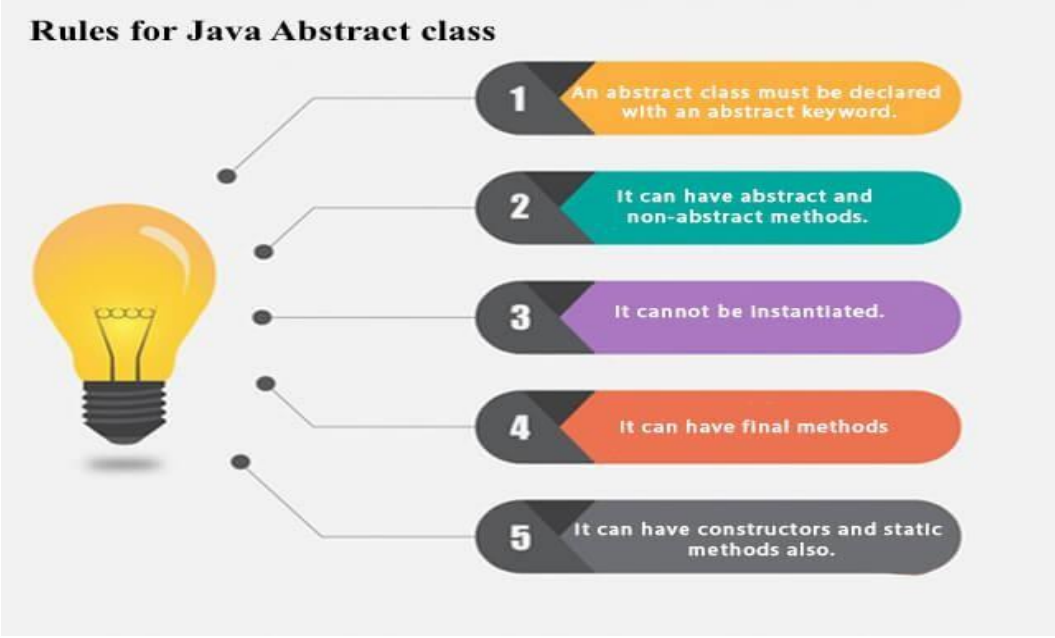
1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.



Example of abstract class

```
1. abstract class A{}
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
1. abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
1. abstract class Bike{
2.   abstract void run();
3. }
4. class Honda4 extends Bike{
5. void run(){System.out.println("running safely");}
6. public static void main(String args[]){
7.  Bike obj = new Honda4();
8.  obj.run();
9. }
10. }
running safely
```

Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
1. abstract class Shape{
2. abstract void draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. class Rectangle extends Shape{
6. void draw(){System.out.println("drawing rectangle");}
7. }
8. class Circle1 extends Shape{
9. void draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. class TestAbstraction1{
13. public static void main(String args[]){
14. Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() m
    ethod
15. s.draw();
16. }
17. }
```

```
drawing circle
```

Another example of Abstract class in java

File: TestBank.java

```
1. abstract class Bank{
2. abstract int getRateOfInterest();
3. }
4. class SBI extends Bank{
5. int getRateOfInterest(){return 7;}
```

```
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

File: TestAbstraction2.java

```
//Example of an abstract class that has abstract and non-abstract methods
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }
//Creating a Child class which inherits Abstract class
 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
//Creating a Test class which calls abstract and non-abstract methods
 class TestAbstraction2{
 public static void main(String args[]){
```

```
14.   Bike obj = new Honda();
15.   obj.run();
16.   obj.changeGear();
17.  }
18. }
```

```
bike is created
running safely..
gear changed
```

> Rule: If there is an abstract method in a class, that class must be abstract.

```
1.  class Bike12{
2.  abstract void run();
3.  }
```

```
compile time error
```

> Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
1.  interface A{
2.  void a();
3.  void b();
4.  void c();
5.  void d();
6.  }
7.
8.  abstract class B implements A{
9.  public void c(){System.out.println("I am c");}
10. }
```

```
11.
12. class M extends B{
13. public void a(){System.out.println("I am a");}
14. public void b(){System.out.println("I am b");}
15. public void d(){System.out.println("I am d");}
16. }
17.
18. class Test5{
19. public static void main(String args[]){
20. A a=new M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}
```

```
Output:I am a
       I am b
       I am c
       I am d
```

UNIT-II

Interface in Java

1. Interface
2. Example of Interface
3. Multiple inheritance by Interface
4. Why multiple inheritance is supported in Interface while it is not supported in case of class.
5. Marker Interface
6. Nested Interface

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

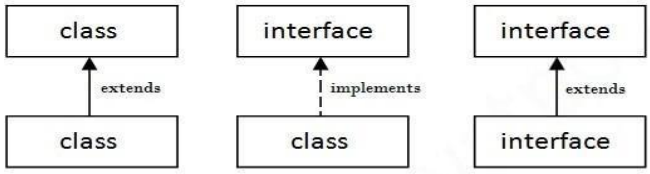
Syntax:

interface **<interface_name>{**

// declare constant fields
// declare methods that abstract
// by default.
}

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example
In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

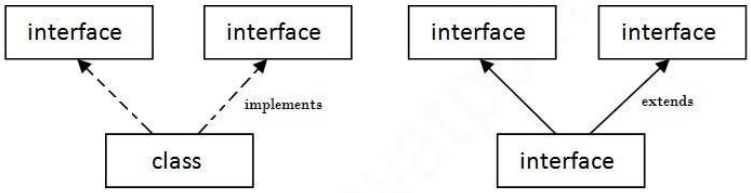
```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Output:

```
Hello
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Test it Now

```
Output:Hello
     Welcome
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

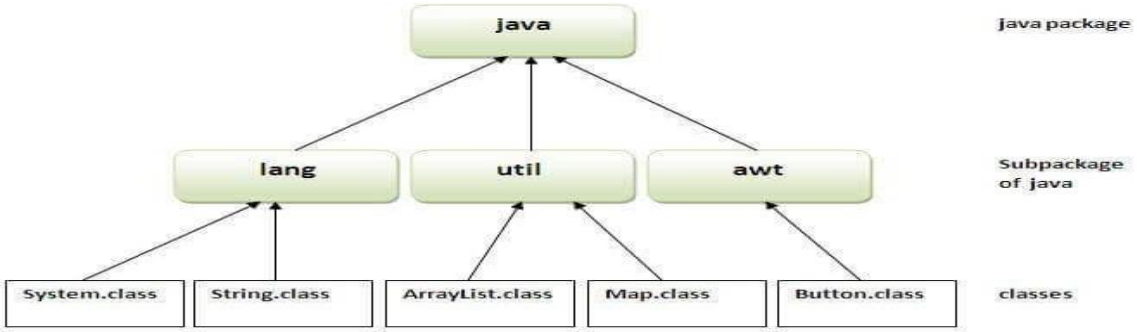
But there are many differences between abstract class and interface that are given below

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.
Package in java can be categorized in two form, built-in package and user-defined package.
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
Here, we will have the detailed learning of creating and using user-defined packages.
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.



Simple example of java package

The package keyword **is used to create a package in java**

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
```

How to compile java package

1. javac -d directory javafilename

   For **example**

1. 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).

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//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

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Access Modifiers in Java

1. Private access modifier
2. Role of private constructor
3. Default access modifier
4. Protected access modifier
5. Public access modifier
6. Access Modifier with Method Overriding

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.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

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

Java.lang package in Java

Provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.

Following are the Important Classes in Java.lang package :

1. **Boolean**: The Boolean class wraps a value of the primitive type boolean in an object.
2. **Byte**: The Byte class wraps a value of primitive type byte in an object.
3. Character – **Set 1**, **Set 2:** The Character class wraps a value of the primitive type char in an object.
4. **Character.Subset**: Instances of this class represent particular subsets of the Unicode character set.

5. **Character.UnicodeBlock**: A family of character subsets representing the character blocks in the Unicode specification.
6. Class – **Set 1**, **Set 2** : Instances of the class Class represent classes and interfaces in a running Java application.
7. ClassLoader: A class loader is an object that is responsible for loading classes.
8. ClassValue: Lazily associate a computed value with (potentially) every type.
9. **Compiler**: The Compiler class is provided to support Java-to-native-code compilers and related services.
10. **Double**: The Double class wraps a value of the primitive type double in an object.
11. **Enum**: This is the common base class of all Java language enumeration types.
12. **Float:** The Float class wraps a value of primitive type float in an object.
13. **InheritableThreadLocal**: This class extends ThreadLocal to provide inheritance of values from parent thread to child thread: when a child thread is created, the child receives initial values for all inheritable thread-local variables for which the parent has values.
14. **Integer** :The Integer class wraps a value of the primitive type int in an object.
15. **Long**: The Long class wraps a value of the primitive type long in an object.
16. Math – **Set 1**, **Set 2**: The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
17. **Number:** The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.
18. **Object**: Class Object is the root of the class hierarchy.
19. **Package**: Package objects contain version information about the implementation and specification of a Java package.
20. **Process**: The ProcessBuilder.start() and Runtime.exec methods create a native process and return an instance of a subclass of Process that can be used to control the process and obtain information about it.
21. **ProcessBuilder**: This class is used to create operating system processes.
22. ProcessBuilder.Redirect: Represents a source of subprocess input or a destination of subprocess output.
23. **Runtime**: Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running.
24. RuntimePermission: This class is for runtime permissions.
25. SecurityManager: The security manager is a class that allows applications to implement a security policy.
26. **Short**: The Short class wraps a value of primitive type short in an object.
27. **StackTraceElement**: An element in a stack trace, as returned by Throwable.getStackTrace().
28. StrictMath- **Set1**, **Set2**: The class StrictMath contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
29. String- **Set1**, **Set2**: The String class represents character strings.
30. StringBuffer: A thread-safe, mutable sequence of characters.
31. StringBuilder: A mutable sequence of characters.
32. **System**: The System class contains several useful class fields and methods.

33. **Thread**: A thread is a thread of execution in a program.
34. **ThreadGroup**: A thread group represents a set of threads.
35. **ThreadLocal**: This class provides thread-local variables.
36. Throwable: The Throwable class is the superclass of all errors and exceptions in the Java language.
37. **Void**: The Void class is an uninstantiable placeholder class to hold a reference to the Class object representing the Java keyword void.

Exception Handling in Java

1. Exception Handling
2. Advantage of Exception Handling
3. Hierarchy of Exception classes
4. Types of Exception
5. Exception Example
6. Scenarios where an exception may occur

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

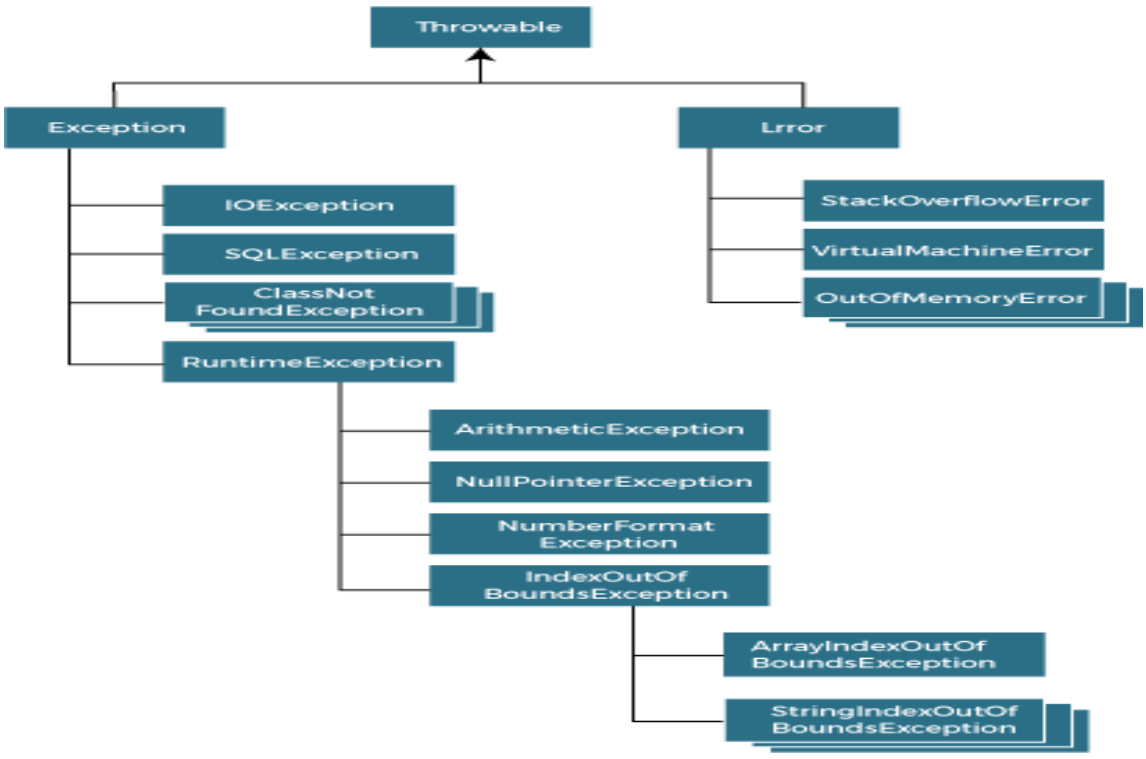
1. statement 1;
2. statement 2;

3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

ava Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
1. public class JavaExceptionExample{
2.   public static void main(String args[]){
3.    try{
4.       //code that may raise exception
```

```
5.        int data=100/0;
6.     }catch(ArithmeticException e){System.out.println(e);}
7.     //rest code of the program
8.     System.out.println("rest of the code...");
9.    }
10. }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
1. try{
2. //code that may throw an exception
3. }catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
1. try{
2. //code that may throw an exception
3. }finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example 1

TryCatchExample1.java

```
public class TryCatchExample1 {

   public static void main(String[] args) {

      int data=50/0; //may throw exception

      System.out.println("rest of the code");

   }

}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example 2

TryCatchExample2.java

```
public class TryCatchExample2 {

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

```
        int data=50/0; //may throw exception
        }
           //handling the exception
        catch(ArithmeticException e)
        {
           System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

UNIT-3

Multithreading and Java I/O

1. Multithreading

Multithreading is a feature in Java that allows concurrent execution of two or more threads. Each thread runs in parallel and can perform different tasks simultaneously. This increases the efficiency and performance of applications, particularly in handling multiple tasks at the same time.

Benefits of Multithreading:

- Resource Sharing: Multiple threads can share the same resources (such as memory, data, files) which is more efficient than multiple processes.

- Responsiveness: Multithreading enhances the responsiveness of applications.

- Utilization of Multiprocessor Architectures: Threads can run on different processors, utilizing the CPU better.

- Economy: Threads are more economical than processes in terms of resource consumption.

2. java. lang. Thread

Thejava. lang. Thread` class is the primary class for creating and managing threads in Java. It provides constructors and methods to create and control threads.

Common Methods:

- start (): Begins the execution of the thread.

- run (): Contains the code that constitutes the new thread.

- sleep (long mills): Causes the thread to sleep for the specified duration.

- join (): Waits for the thread to die.

- is Alive (): Tests if the thread is alive.

- get Priority () / set Priority (int priority): Gets or sets the priority of the thread.

3. The Main Thread

The main thread is the initial thread of execution in a Java application. It is created automatically when the application starts and can create other threads.

Example:

java

```
public class MainThreadExample {
    public static void main (String [] args) {
        Thread main Thread = Thread.currentThread();
        System.out.println("Main thread: " + mainThread.getName());
    }
}
```

4. Creation of New Threads

There are two main ways to create new threads in Java:

1. Extending the Thread class:

java

```
class MyThread extends Thread {
    public void run () {
        System.out.println("Thread  running");
    }
}

public class Test {
    public static void main (String [] args) {
        MyThread t1 = new MyThread ();
        t1.start();
    }
}
```

2. Implementing the Runnable interface:

java

```
class MyRunnable implements Runnable {
    public void run () {
        System.out.println("Thread  running");
    }
}

public class Test {
    public static void main (String [] args) {
        Thread t1 = new Thread (new MyRunnable ());
        t1.start();
    }
}
```

5. Thread Priority

Each thread has a priority that helps the thread scheduler determine the order of thread execution. Priorities are integers ranging from `Thread.MIN_PRIORITY` (1) to `Thread.MAX_PRIORITY` (10), with `Thread.NORM_PRIORITY` (5) as the default.

Example:

java

```
Thread t1 = new Thread (new MyRunnable ());
t1.setPriority(Thread.MAX_PRIORITY);
t1.start();
```

6. Multithreading using isAlive () and join ()

isAlive (): Checks if a thread has finished execution.

join (): Waits for a thread to terminate.

Example:

java

```
Thread t1 = new Thread (new MyRunnable ());
t1.start();

try {
    t1.join();
} catch (InterruptedException e) {
e. printStackTrace ();
}

if (t1. isAlive ()) {
    System.out.println("Thread is still running");
} else {
    System.out.println("Thread has finished");
}
```

7. Synchronization

Synchronization is used to control the access of multiple threads to shared resources. The synchronized` keyword can be applied to methods or blocks to prevent thread interference and ensure data consistency.

Example:

java

```
class Counter {
    private int count = 0;

    public synchronized void increment () {
        count++;
```

```
    }

    public int getCount () {
        return count;
    }
}

public class Test {
    public static void main (String [] args) {
        Counter counter = new Counter ();

        Thread t1 = new Thread (() -> {
            for (int i = 0; i< 1000; i++) {
counter. increment ();
            }
        });

        Thread t2 = new Thread (() -> {
            for (int i = 0; i< 1000; i++) {
counter. increment ();
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2. join ();
        } catch (InterruptedException e) {
```

```
e. printStackTrace ();
        }

        System.out.println("Count: " + counter. getCount ());
    }
}
```

8. Suspending and Resuming Threads

The methods `suspend () ` and `resume () ` are deprecated due to their unsafe nature. Instead, modern approaches use `wait () ` and `notify () ` for thread communication.

Example:

```
java
class SuspendResumeExample {
    public synchronized void work () throws InterruptedException {
wait (); // Suspends the thread
    }

    public synchronized void resume Work () {
notify (); // Resumes the thread
    }
}
```

9. Communication Between Threads

Threads can communicate by using `wait () `, `notify () `, and `notify All ()` methods. These methods are used to implement the producer-consumer problem and other scenarios requiring synchronization and communication.

Example:

```
java
class Shared Resource {
    private int value = 0;
    private boolean valueSet = false;

    public synchronized void produce (int value) throws InterruptedException {
        while (valueSet) {
wait ();
        }
this. Value = value;
        valueSet = true;
notify ();
    }

    public synchronized int consume () throws InterruptedException {
        while (! valueSet) {
wait ();
        }
        valueSet = false;
notify ();
        return value;
    }
}

public class Test {
    public static void main (String [] args) {
Shared Resource resource = new SharedResource ();

        Thread producer = new Thread (() -> {
            try {
```

```
resource. Produce (1);
            } catch (InterruptedException e) {
e. printStackTrace ();
            }
        });

        Thread consumer = new Thread (() -> {
            try {
                System.out.println(resource. Consume ());
            } catch (InterruptedException e) {
e. printStackTrace ();
            }
        });

producer. Start ();
consumer. Start ();
    }
}
```

10. Input/Output: Reading and Writing Data

Java provides the `java.io` package for input and output operations. Key classes include `FileInputStream`, `FileOutputStream`, `BufferedReader`, `Buffered Writer`, `ObjectInputStream`, and `ObjectOutputStream`.

Reading and Writing Text Files

- Reading a File:

```
java
import java.io. *;

public class ReadFile {
```

```
    public static void main (String [] args) {
        try (BufferedReader reader = new BufferedReader (new File Reader("input.txt"))) {
            String line;
            while ((line = reader. read Line ()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
e. printStackTrace ();
        }
    }
}
```

- Writing to a File:

```
java
import java.io. *;

public class WriteFile {
    public static void main (String [] args) {
        try (Buffered Writer = new BufferedWriter (new FileWriter("output.txt"))) {
writer. Write ("Hello, World!");
        } catch (IOException e) {
e. printStackTrace ();
        }
    }
}
```

Reading and Writing Binary Data

- Reading a File:

java

```
import java.io. *;

public class ReadBinaryFile {
   public static void main (String [] args) {
      try (FileInputStream fis = new FileInputStream("input.bin")) {
         int content;
         while ((content = fis. read ()) != -1) {
System.out.println((char) content);
         }
      } catch (IOException e) {
e. printStackTrace ();
      }
   }
}
```

- Writing to a File:

java

```
import java.io. *;

public class WriteBinaryFile {
   public static void main (String [] args) {
      try (FileOutputStream fos = new FileOutputStream("output.bin")) {
fos. write ("Hello, World! “getBytes ());
      } catch (IOException e) {
e. printStackTrace ();
      }
   }
}
```

Object Serialization

- Writing an Object:

```java
import java.io. *;

class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    String name;
    int age;

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

public class Write Object {
    public static void main (String [] args) {
        Person person = new Person ("John", 30);

        try (ObjectOutputStream oos = new ObjectOutputStream (new
FileOutputStream("person.obj"))) {
oos. write Object(person);
        } catch (IOException e) {
e. printStackTrace ();
        }
    }
}
```

- Reading an Object:

```
java

import java.io. *;


public class ReadObject {

   public static void main (String [] args) {

      try (ObjectInputStream ois = new ObjectInputStream (new
FileInputStream("person.obj"))) {

         Person person = (Person) ois. readObject();

         System.out.println("Name: " + person.name + ", Age: " + person. Age);

      } catch (IOException | ClassNotFoundException e) {

e. printStackTrace ();

      }

   }

}
```

These notes cover the essential aspects of multithreading and I/O in Java, providing a solid foundation for understanding and implementing these concepts

APPLETS NOTES

1. Introduction to Applets

Applets are small Java programs that can be embedded in web pages and run in a web browser or an applet viewer. They are typically used to create dynamic and interactive web applications.

Key Features:

- Embedded in HTML: Applets are embedded in HTML documents using the `<applet>` or `<object>` tag.

- Run in Browser: They run inside a web browser with Java plugin or an applet viewer.

- Sandboxed Environment: Applets run in a restricted environment (sandbox) to ensure security.

2. Applet Lifecycle

An applet goes through a series of states from initialization to destruction. These states are managed by the browser or applet viewer.

Lifecycle Methods:

1. init (): Called once when the applet is first loaded. Used for initialization.

2. start (): Called each time the applet is started or revisited.

3. stop(): Called each time the applet is stopped or the user navigates away.

4. destroy(): Called once when the applet is being removed from memory. Used for cleanup.

5. paint (Graphics g): Called whenever the applet needs to repaint itself.

Example:

java

```
import java. applet. Applet;
import java.awt. Graphics;

public class MyApplet extends Applet {
    public void init () {
        // Initialization code
    }

    public void start () {
        // Start or resume execution
    }

    public void stop () {
        // Suspend execution
    }
```

```
    public void destroys () {
        // Cleanup before unloading
    }

    public void paint (Graphics g) {
drawstring("Hello, Applet!", 20, 20);
    }
}
```

3. HTML for Embedding Applets

To embed an applet in an HTML document, use the `<applet>` or `<object>` tag.

Using `<applet>` Tag:

```
`html
<applet code="MyApplet.class" width="300" height="300">
    Your browser does not support Java Applets.
</applet>
```

Using `<object>` Tag:

```
html
<object classid="java: MyApplet.class" width="300" height="300">
    Your browser does not support Java Applets.
</object>
```

4. Applet Security

Applets run in a sandboxed environment to prevent unauthorized access to the system. This includes restrictions on file I/O, network access, and system properties. However, signed applets can request permissions for certain operations.

Key Security Features:

- Sandbox Restrictions: Limits access to local file systems, network connections, and system resources.

- Code Signing: Applets can be digitally signed to request additional permissions.

- Security Manager: Controls what operations an applet can perform.

5. Graphics in Applets

Applets use the AWT (Abstract Window Toolkit) library for drawing graphics. The `paint ()` method is used to render graphics on the applet's surface.

Common Graphics Methods:

- drawString (String str, int x, int y): Draws a string at the specified coordinates.

- draw Line (int x1, int y1, int x2, int y2): Draws a line between two points.

- drawRect (int x, int y, int width, int height): Draws a rectangle.

- draw Oval (int x, int y, int width, int height): Draws an oval.

Example:

```
java
import java. applet. Applet;
import java.awt. Graphics;

public class GraphicsApplet extends Applet {
    public void paint (Graphics g) {
g. drawString ("Hello, Graphics!", 50, 50);
g. draw Line (50, 60, 150, 60);
drawRect (50, 70, 100, 50);
g. draw Oval (50, 130, 100, 50);
    }
}
```

6. Handling Events in Applets

Applets can handle user input and interaction through event handling mechanisms provided by AWT and Swing libraries.

Common Event Classes:

- Mouse Event: Handles mouse actions like clicks, movements.
- Key Event: Handles keyboard actions.
- Action Event: Handles actions performed by buttons and other components.

Example:

```
java
import java. applet. Applet;
import java.awt. Graphics;
import java.awt. event. Mouse Event;
import java.awt. event. Mouse Listener;

public class Event Applet extends Applet implements Mouse Listener {
    private String message = "";

    public void init () {
        addMouseListener(this);
    }

    public void paint (Graphics g) {
g. drawString (message, 20, 20);
    }

    public void mouse Clicked (Mouse Event e) {
        message = "Mouse clicked at (" + egret () + ", " + egret () + ")";
repaint ();
```

```
    }

    // Other MouseListener methods
    public void mouseEntered (MouseEvent e) {}
    public void mouse Exited (MouseEvent e) {}
    public void mouse Pressed (MouseEvent e) {}
    public void mouse Released (MouseEvent e) {}
}
```

7. Applet Parameters

Applets can accept parameters from the HTML file. These parameters are passed as key-value pairs using the `<param>` tag.

Example:

```html
<applet code="ParamApplet.class" width="300" height="300">
<param name="message" value="Hello, Param!">
</applet>
```

Retrieving Parameters:

```
java
import java. applet. Applet;
import java.awt. Graphics;

public class Param Applet extends Applet {
    private String message;

    public void init () {
        message = get Parameter("message");
        if (message == null) {
```

```
            message = "No message.";
        }
    }

    public void paint (Graphics g) {
g. drawString (message, 20, 20);
    }
}
```

8. Advantages and Disadvantages of Applets

Advantages:

- Cross-Platform: Applets can run on any platform with a compatible Java runtime environment.

- Interactive: Provide a rich, interactive user experience within web pages.

- Security: Run in a sandboxed environment, minimizing security risks.

Disadvantages:

- Browser Support: Limited support in modern browsers as they phase out NPAPI plugin support.

- Performance: May not perform well for complex applications.

- Deployment: Requires users to have Java installed and configured correctly.

9. Alternatives to Applets

With the decline in applet support in modern browsers, other technologies have become popular for web applications:

- JavaScript and HTML5: For rich, interactive web applications without the need for plugins.

- Java Web Start: For launching full-featured Java applications directly from the web.

- Flash and Silverlight: (Now deprecated) Were alternatives for interactive web content.

- Modern Java Frameworks: Spring Boot, Angular, React for building modern web applications.

These notes provide a comprehensive overview of Java applets, their lifecycle, graphics handling, event management, security, and modern alternatives.

UNIT-4

Introduction to Event Handling

Event handling in Java refers to the mechanism that allows a program to respond to events such as user actions (e.g., mouse clicks, key presses). The Java event handling mechanism is built around three main components:

1. **Event Source**: The object that generates an event.
2. **Event Object**: Encapsulates the event information.
3. **Event Listener**: The object that receives and handles the event.

The process of event handling can be broken down into the following steps:

1. An event source generates an event.
2. The event is passed to an event listener.
3. The event listener processes the event and responds accordingly.

Event Delegation Model

The Event Delegation Model is a design pattern used in Java to handle events. Instead of each component handling its own events, a central component (the listener) handles events for multiple sources. This approach simplifies event handling by separating the event generation from event processing.

Key aspects of the Event Delegation Model:

- **Event Source**: The component that generates the event (e.g., a button).
- **Event Listener**: An interface that receives and processes the event.
- **Event Object**: Encapsulates information about the event.

The model relies on registering event listeners with event sources. When an event occurs, the source forwards the event to the appropriate listener.

java.awt.event Description

The java.awt.event package provides a set of interfaces and classes for dealing with various types of events generated by AWT components.

Key classes and interfaces in java.awt.event:

- **ActionEvent**: Generated by a component (like a button) when an action occurs.
- **AdjustmentEvent**: Generated by adjustable components like scrollbars.
- **ComponentEvent**: Generated when a component is hidden, shown, moved, or resized.
- **ContainerEvent**: Generated when a component is added or removed from a container.
- **FocusEvent**: Generated when a component gains or loses keyboard focus.

- **ItemEvent**: Generated by components that can be selected or deselected (e.g., checkboxes).
- **KeyEvent**: Generated by a component when a key is pressed, released, or typed.
- **MouseEvent**: Generated when the mouse is clicked, pressed, released, moved, or dragged.
- **WindowEvent**: Generated when a window is opened, closed, activated, deactivated, iconified, or deiconified.

Sources of Events

Sources of events are typically GUI components such as buttons, text fields, windows, etc. These components generate events when interacted with by the user. Common sources of events include:

- **Buttons**: Generate ActionEvent when clicked.
- **TextFields**: Generate ActionEvent when enter is pressed.
- **Windows**: Generate WindowEvent for various window operations.
- **Mouse**: Generates MouseEvent for mouse actions.

Event Listeners

Event listeners are interfaces that must be implemented to handle specific types of events. Some common event listeners include:

- **ActionListener**: Handles ActionEvent.
- **AdjustmentListener**: Handles AdjustmentEvent.
- **ComponentListener**: Handles ComponentEvent.
- **FocusListener**: Handles FocusEvent.
- **ItemListener**: Handles ItemEvent.
- **KeyListener**: Handles KeyEvent.
- **MouseListener**: Handles MouseEvent.
- **WindowListener**: Handles WindowEvent.

To handle an event, a listener must:

1. Implement the appropriate listener interface.
2. Register with an event source to receive notifications.

Adapter Classes

Adapter classes provide default implementations of event listener interfaces. They are useful when you want to handle only a subset of the events defined in an interface. For example:

- **MouseAdapter**: Provides default implementations for MouseListener methods.
- **KeyAdapter**: Provides default implementations for KeyListener methods.
- **WindowAdapter**: Provides default implementations for WindowListener methods.

Using an adapter class, you can override only the methods you need, instead of implementing all methods in the interface.

Inner Classes

Inner classes in Java can be used to implement event listeners. They are useful for keeping the event handling code close to the GUI code. There are two types of inner classes that are often used for event handling:

- **Member Inner Classes**: Defined at the member level of a class.
- **Anonymous Inner Classes**: Defined and instantiated in a single expression. They are often used for short, one-off event handling implementations.

Example of an anonymous inner class for a button's action listener:

```
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      // handle button click
   }
});
```

Abstract Window Toolkit (AWT)

Why AWT? AWT is a part of the Java Foundation Classes (JFC), which provides a rich set of GUI components. AWT components are heavyweight, which means they are associated with their native screen resources. AWT was the original Java GUI toolkit and provides a platform-independent way to create graphical user interfaces.

java.awt Package The `java.awt` package contains classes for creating user interfaces and for painting graphics and images. Key components include:

- **Component**: The base class for all AWT components.
- **Container**: A subclass of Component that can contain other components.

Components and Containers Components are the basic building blocks of an AWT application. Containers are special components that can hold other components, creating a hierarchical structure.

Common AWT Components

1. **Button**
   - A Button is a GUI component that triggers an action when clicked.
   - Example:

     ```
     Button b = new Button("Click Me");
     ```

2. **Label**
   - A Label is a non-editable text component used to display a short string or an image.
   - Example:

     ```
     Label l = new Label("Hello, World!");
     \
     ```

3. **Checkbox**
    - A Checkbox is a component that can be either checked or unchecked.
    - Example:

```
Checkbox c = new Checkbox("Accept Terms");
```

4. **Radio Buttons**
    - Radio buttons are used for selecting one option from a group.
    - Managed using the `CheckboxGroup` class.
    - Example:

```
CheckboxGroup group = new CheckboxGroup();
Checkbox r1 = new Checkbox("Option 1", group, false);
Checkbox r2 = new Checkbox("Option 2", group, true);
```

5. **List Boxes**
    - A List is a component that allows the user to select one or more items from a list.
    - Example:

```
List list = new List();
list.add("Item 1");
list.add("Item 2");
```

6. **Choice Boxes**
    - A Choice is a drop-down list that allows the user to select an item from a list.
    - Example:

```
Choice choice = new Choice();
choice.add("Option 1");
choice.add("Option 2");
```

7. **Text Field and Text Area**
    - **TextField**: A single-line text input component.

```
TextField tf = new TextField();
```

    - **TextArea**: A multi-line text input component.

```
TextArea ta = new TextArea();
```

Container Classes

Containers are components that can hold other components. Key container classes include:

- **Panel**: A generic container for organizing components.
- **Frame**: A top-level window with a title and a border.
- **Applet**: A container for creating applets (deprecated in recent Java versions).

Layout Managers

Layout managers control the size and position of components within a container. Common layout managers include:

- **FlowLayout**: Arranges components in a left-to-right flow.
- **BorderLayout**: Arranges components in five regions: north, south, east, west, and center.
- **GridLayout**: Arranges components in a grid of cells.

Menu

AWT provides classes to create menus:

- **MenuBar**: The menu bar.
- **Menu**: A pull-down menu.
- **MenuItem**: An item in a menu.

Scroll Bar

AWT provides the Scrollbar class to create scrollbars.

- Example:

  ```
  Scrollbar s = new Scrollbar();
  ```

Swing Introduction

Swing is a part of the Java Foundation Classes (JFC) and provides a rich set of GUI components. Swing components are lightweight and written entirely in Java, providing a more flexible and powerful approach to GUI development than AWT.

Key Swing Components

1. **JFrame**
   - A top-level container used to create a window.
   - Example:

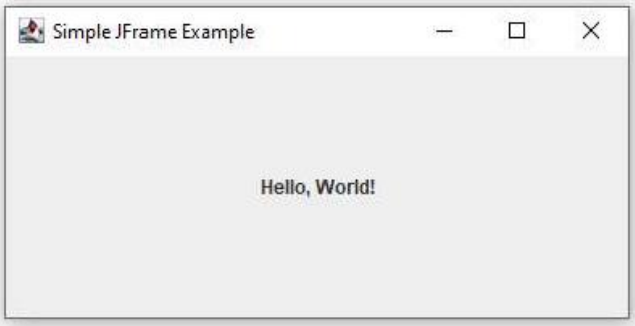
     ```
     JFrame frame = new JFrame("My Frame");
     ```

Program

```
package com.example;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;

public class SimpleJFrame {
    public static void main(String[] args) {
        // Use SwingUtilities.invokeLater to ensure thread safety
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                // Create the JFrame instance
                JFrame frame = new JFrame("Simple JFrame Example");
```

```
            // Create a JLabel with text
            JLabel label = new JLabel("Hello, World!", JLabel.CENTER);

            // Add the label to the frame
            frame.add(label);

            // Set the frame size
            frame.setSize(400, 200);

            // Exit the application when the frame is closed
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            // Make the frame visible
            frame.setVisible(true);
        }
    });
  }
}
```

```java
package com.example;

import javax.swing.JFrame;

public class SimpleJFrame {
    public static void main(String[] args) {
        // Use SwingUtilities.invokeLater to ensure thread safety
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                // Create the JFrame instance
                JFrame frame = new JFrame("Simple JFrame Example");

                // Create a JLabel with text
                JLabel label = new JLabel("Hello, World!", JLabel.CENTER);

                // Add the label to the frame
                frame.add(label);

                // Set the frame size
                frame.setSize(400, 200);

                // Exit the application when the frame is closed
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                // Make the frame visible
                frame.setVisible(true);
            }
        });
    }
}
```



2. **JApplet**
    - A Swing-based applet.
    - Example:

```
public class MyApplet extends JApplet {
    public void init() {
        // initialization code
    }
}
```

3. **JPanel**
    - A generic container for holding components.
    - Example:

```
JPanel panel = new JPanel();
```

Components in Swing

Swing provides a wide range of components including:

- **JButton**
- **JLabel**
- **JCheckBox**
- **JRadioButton**
- **JList**
- **JComboBox**
- **JTextField**
- **JTextArea**

Layout Managers in Swing

Swing supports all the layout managers available in AWT, and also provides additional ones like:

- **BoxLayout**: Organizes components either horizontally or vertically.
- **GroupLayout**: Allows for placing components in groups, making it easier to align them.

Specialized Swing Components

1. **JList and JScrollPane**
    - **JList**: Displays a list of items for selection.

```
JList<String> list = new JList<>(new String[] {"Item 1", "Item 2"});
```

    - **JScrollPane**: Adds scroll functionality to components.

```
JScrollPane scrollPane = new JScrollPane(list);
```

2. **JSplitPane**
    - Divides two components with a resizable divider.
    - Example:

```
JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, component1, component2);
```

3. **JTabbedPane**
    - Organizes components into tabs.
    - Example:

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab("Tab 1", component1);
tabbedPane.addTab("Tab 2", component2);
```

4. **JDialog**
    - A top-level window for taking input from the user.
    - Example:

```
JDialog dialog = new JDialog(frame, "Dialog", true);
```

5. **Pluggable Look and Feel**
    - Swing allows changing the look and feel of components.
    - Example:

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

1.Program on JLabel

```
package swing;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloWorldSwing {
   public static void main(String[] args) {
      // Create a new frame
      JFrame frame = new JFrame("HelloWorldSwing");

      // Create a label with text
      JLabel label = new JLabel("Hello, World!", JLabel.CENTER);

      // Add the label to the frame
      frame.add(label);

      // Set the frame size
      frame.setSize(300, 200);

      // Exit the application when the frame is closed
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        // Make the frame visible
        frame.setVisible(true);
    }
}
```

JCheckBox Code:

```
package com.example;

import javax.swing.JFrame;
import javax.swing.JCheckBox;
import javax.swing.JLabel;
import javax.swing.JPanel;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.BorderLayout;

public class SimpleJCheckBox {
    public static void main(String[] args) {
        // Create a new frame
        JFrame frame = new JFrame("JCheckBox Example");

        // Create a label to display the state of checkboxes
        JLabel label = new JLabel("Checkbox State: ");

        // Create two checkboxes
        JCheckBox checkBox1 = new JCheckBox("Option 1");
        JCheckBox checkBox2 = new JCheckBox("Option 2");

        // Add an item listener to checkBox1
        checkBox1.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                updateLabel(label, checkBox1, checkBox2);
            }
        });

        // Add an item listener to checkBox2
        checkBox2.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                updateLabel(label, checkBox1, checkBox2);
            }
        });

        // Create a panel to hold checkboxes
        JPanel panel = new JPanel();
        panel.add(checkBox1);
        panel.add(checkBox2);

        // Add the panel and label to the frame
```

```
        frame.add(panel, BorderLayout.CENTER);
        frame.add(label, BorderLayout.SOUTH);

        // Set the frame size
        frame.setSize(300, 200);

        // Exit the application when the frame is closed
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Make the frame visible
        frame.setVisible(true);
    }

    // Update the label to reflect the state of the checkboxes
    private static void updateLabel(JLabel label, JCheckBox checkBox1, JCheckBox checkBox2) {
        StringBuilder sb = new StringBuilder("Checkbox State: ");
        sb.append(checkBox1.getText()).append(" is ").append(checkBox1.isSelected() ? "selected" :
"not selected");
        sb.append(", ");
        sb.append(checkBox2.getText()).append(" is ").append(checkBox2.isSelected() ? "selected" :
"not selected");
        label.setText(sb.toString());
    }
}
```

output