PR DEGREE GOVERNMENT COLLEGE(A)

Object Oriented Programming through Java

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.

Applications of OOPs:

Object-oriented programming (OOP) offers a robust framework for software development by organizing code into objects that interact with each other. Here are some common applications of OOP:

1. Software Development: OOP is widely used in software development across various domains such as web development, mobile app development, game development, and enterprise software development.
2. Graphical User Interface (GUI) Development: OOP is particularly useful in GUI development because it allows developers to create reusable components (objects) that can be easily manipulated and interacted with in graphical applications.
3.Simulation and Modeling: OOP is utilized in simulation and modeling applications where real-world systems are simulated using objects to represent the various components and their interactions.
4. Database Systems: Object-oriented databases (OODBMS) use OOP concepts to model data as objects, providing a more natural way to represent complex relationships and structures within data.
5.Artificial Intelligence and Machine Learning: OOP principles are applied in AI and ML systems to organize code into reusable and scalable components, making it easier to manage and maintain complex systems.
6.Web Development: OOP is commonly used in web development frameworks such as Django (Python) and Ruby on Rails (Ruby) to structure code into reusable classes and objects, facilitating easier development and maintenance of web applications.
7.Game Development: OOP is extensively used in game development to represent game entities, behaviors, and interactions. Game engines like Unity and Unreal Engine rely heavily on OOP principles to create interactive and immersive gaming experiences.
8.Embedded Systems: OOP is applied in embedded systems programming to organize code into modular and reusable components, improving code readability, scalability, and maintainability.
9.Financial Systems: OOP is used in financial systems to model complex financial instruments, transactions, and calculations, providing a structured and maintainable approach to handling financial data and processes.
10.Networking and Systems Programming: OOP is employed in networking and systems programming to model network protocols, devices, and services as objects, simplifying the development and management of networked systems.

Features of JAVA:

Java is a powerful and versatile programming language with a rich set of features that contribute to its popularity and widespread use. Here are some key features of Java:

1.Platform Independence: Java programs can run on any device or platform that supports the Java Virtual Machine (JVM), making them platform independent. This is achieved through the concept of "Write Once, Run Anywhere" (WORA).
2.Object-Oriented: Java is an object-oriented programming language, which means it supports concepts like encapsulation, inheritance, and polymorphism, allowing for modular and reusable code.
3.Simple and Familiar Syntax: Java's syntax is similar to C and C++, making it easy for developers to learn and understand, especially for those with a background in other programming languages.
4.Automatic Memory Management: Java uses a garbage collector to automatically manage memory allocation and deallocation, relieving the programmer from manual memory management tasks like in languages such as C or C++.
5.Robust and Secure: Java's strong type system, exception handling mechanisms, and security features (such as the Java Security Manager) help create robust and secure applications, protecting against common vulnerabilities like buffer overflows and pointer manipulation.
6.Multi-threading Support: Java provides built-in support for multithreading, allowing concurrent execution of multiple tasks within a single program. This feature is essential for developing scalable and responsive applications.
7.Rich Standard Library: Java comes with a comprehensive standard library (Java API) that provides pre-built classes and methods for common programming tasks, ranging from data structures and networking to I/O operations and GUI development.
8.Dynamic and Extensible: Java supports dynamic loading of classes and dynamic compilation, enabling features like reflection and dynamic proxies. Additionally, Java's modular system introduced in Java 9 allows for building and maintaining large-scale applications more effectively.
9.High Performance: Although Java is not as low-level as languages like C or C++, it still offers high performance through features like just-in-time (JIT) compilation, which translates Java bytecode into native machine code at runtime for execution.
10.Community and Ecosystem: Java has a vibrant and active community of developers, along with a vast ecosystem of third-party libraries, frameworks, and tools that enhance productivity and enable developers to tackle a wide range of tasks efficiently.

.

Naming Conventions in Java :

In java, it is good practice to name class, variables, and methods name as what they are supposed to do instead of naming them randomly. Below are some naming conventions of the java programming language. They must be followed while developing software in java for good maintenance and readability of code. Java uses CamelCase as a practice for writing names of methods, variables, classes, packages, and constants.

Camel's case in java programming consists of compound words or phrases such that each word or abbreviation begins with a capital letter or first word with a lowercase letter, rest all with capital. Here in simpler terms, it means if there are two:

In package, everything is small even while we are combining two or more words in java.
In constants, we do use everything as uppercase and only '_' character is used even if we are combining two or more words in java.

Type 1: Classes and Interfaces

Class names should be nouns, in mixed cases with the first letter of each internal word capitalized. Interfaces names should also be capitalized just like class names.
Use whole words and must avoid acronyms and abbreviations.
Classes: *class Student {}*
class Integer {}
class Scanner {}

Interfaces: *Runnable*
Remote
Serializable

Type 2: Methods

Methods should be verbs, in mixed case with the first letter lowercase and with the first letter of each internal word capitalized.
public static void main (String [] args) {}
As the name suggests the method is supposed to be primarily method which indeed it is as main() method in java is the method from where the program begins its execution.

Type 3: Variables

Variable names should be short yet meaningful.
Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed.

- ✓ Should be mnemonic i.e, designed to indicate to the casual observer the intent of its use.
- ✓ One-character variable names should be avoided except for temporary variables.
- ✓ Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

Int [] marks ;
double double answer,

As the name suggests, one stands for marks while the other for an answer be it of any e do not mind.

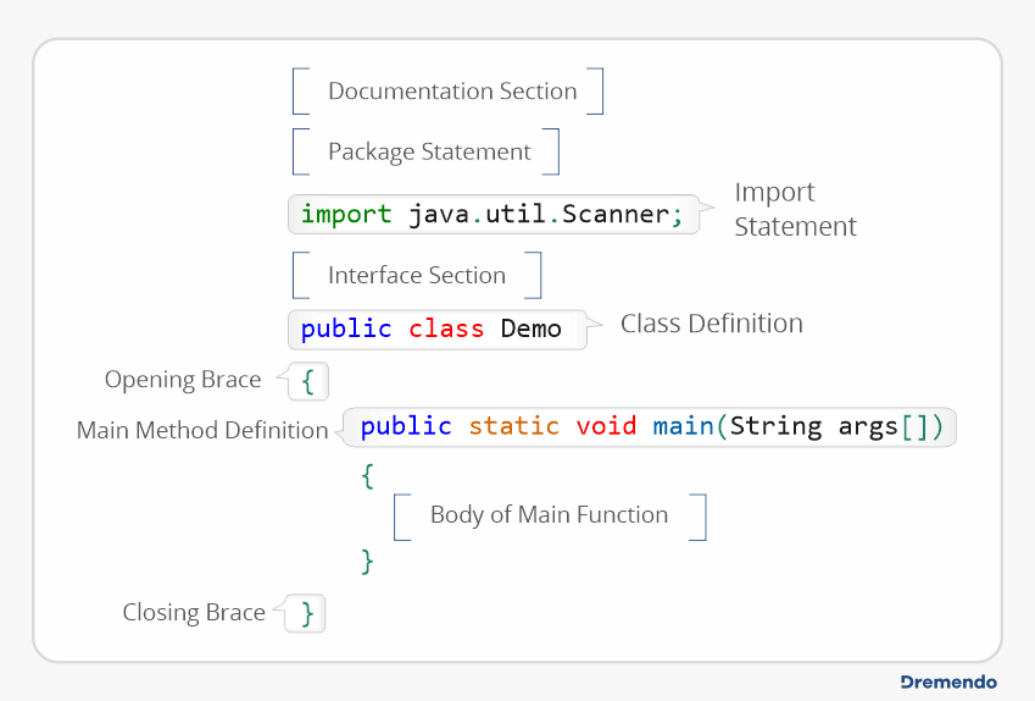
Type 4: Constant variables

Should be all uppercase with words separated by underscores ("_").
There are various constants used in predefined classes like Float, Long, String etc.
num = PI;

Type 5: Packages

The prefix of a unique package name is always written in all-lowercase ASCII letters and should be
Subsequent components of the package name vary according to an organization's own internal naming conventions.
java.util.Scanner ;
java.io.*;

Simple Structure of a Java Program



Example: This is how a simple basic structure of a Java program looks like.

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

Documentation Section: It is used to improve the readability of the program. It consists of comments in Java which include basic information such as the method's usage or functionality to make it easier for the programmer to understand it while reviewing or debugging the code. This statement is optional.

Package Statement: There is a provision in Java that allows us to declare our classes in a collection called package. There can be only one package statement in a Java program and it has to be at the beginning of the code before any class or interface declaration. This statement is optional.

Import Statement: Many predefined classes are stored in packages in Java, an import statement is used to refer to the classes stored in other packages. An import statement is always written after the package statement but it has to be before any class declaration.

Interface Section: This section is used to specify an interface in Java. It is an optional section which is mainly used to implement Multiple Inheritance in Java. An interface is a lot similar to a class in Java but it contains only constants and method declarations.

Class Definition: A Java program may contain several class definitions, classes are an essential part of any Java program. Every program in Java will have at least one class with the main method. The class that contains the main method must be declared as public. For example:

- **public class Test** - This creates a class called Test. You should make sure that the class name starts with a capital letter, and the public word means it is accessible from any other classes.

Braces (Both Opening and Closing Brace) :The curly braces are used to group all the commands together. To make sure that the commands belong to a class or a method.{}

Main Method Definition: *In Java, the main method is treated as the entry point of the program, The main method is called by the operating system when the user runs the program*.

For Example: public static void main(String args[])

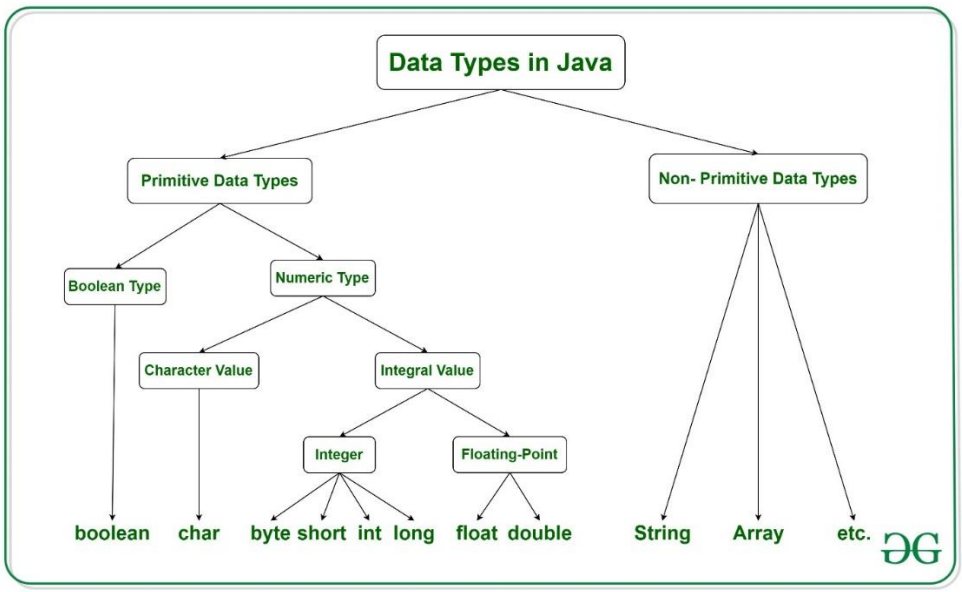
- **public -** When the main method is declared public, it means that it can be used outside of the declared class as well.
- **static -** The word static means that we want to access a method without making object of the class within which the method is declared. We call the main method without creating any objects.
- **void** - The word void indicates that it does not return any value. The main method is declared as void because it does not return any value.
- **main** - The main is the method, which is an essential part of any Java program.
- **String args[]** - It is an array where each element is a string, which is named as args. If you run the Java code through a console, you can pass the input parameter. The main() takes it as an input.

Data Types in Java: Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated.

- ✓ Primitive Data Type
- ✓ Non-Primitive Data Type

Primitive Data Type: Primitive data are only single values and have no special capabilities. There are 8 primitive data types. They are depicted below in tabular format below as follows :such as Boolean, char, int, short, byte, long, float, and double

Non-Primitive Data Type or Object Data type: The Reference Data Types will contain a memory address of variable values because the reference types won't store the variable value directly in memory. They are strings, objects, arrays, etc. such as String, Array, etc.



Data Type	Default Value	Default size	Range
byte	0	1 byte or 8 bits	-128 to 127
short	0	2 bytes or 16 bits	-32,768 to 32,767
int	0	4 bytes or 32 bits	2,147,483,648 to 2,147,483,647
long	0	8 bytes or 64 bits	9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	0.0f	4 bytes or 32 bits	1.4e-045 to 3.4e+038

double	0.0d	8 bytes or 64 bits	4.9e-324 to 1.8e+308
char	'\u0000'	2 bytes or 16 bits	0 to 65536
boolean	FALSE	1 byte or 2 bytes	0 or 1

Java Operators: Operators are symbols that perform operations on variables and values. For example, `+` is an operator used for addition, while `*` is also an operator used for multiplication.

Operators in Java can be classified into 6 types:

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Unary Operators
6. Bitwise Operators

1. Java Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

Here, the **+** operator is used to add two variables **a** and **b.** Similarly, there are various other arithmetic operators in Java.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

2. Java Assignment Operators

Assignment operators are used in Java to assign values to variables. For example,

Here, **=** is the assignment operator. It assigns the value on its right to the variable on its left. That is, **5** is assigned to the variable age.
Let's see some more assignment operators available in Java.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

3. Java Relational Operators
Relational operators are used to check the relationship between two operands. For example,
Here, < operator is the relational operator. It checks if a is less than b or not.
It returns either true or false.

Operator	Description	Example
==	Is Equal To	3 == 5 returns **false**
!=	Not Equal To	3 != 5 returns **true**
>	Greater Than	3 > 5 returns **false**
<	Less Than	3 < 5 returns **true**
>=	Greater Than or Equal To	3 >= 5 returns **false**
<=	Less Than or Equal To	3 <= 5 returns **true**

4. Java Logical Operators
Logical operators are used to check whether an expression is true or false. They are used in decision making.

Operator	Example	Meaning
`&&` (Logical AND)	expression1 **&&** expression2	`true` only if both `expression1` and `expression2` are `true`
`||` (Logical OR)	expression1 **||** expression2	`true` if either `expression1` or `expression2` is `true`
`!` (Logical NOT)	**!**expression	`true` if `expression` is `false` and vice versa

5. Java Unary Operators

Unary operators are used with only one operand. For example, `++` is a unary operator that increases the value of a variable by **1**. That is, `++5` will return **6**.

Different types of unary operators are:

Operator	Meaning
`+`	**Unary plus**: not necessary to use since numbers are positive without using it
`-`	**Unary minus**: inverts the sign of an expression
`++`	**Increment operator**: increments value by 1
`--`	**Decrement operator**: decrements value by 1
`!`	**Logical complement operator**: inverts the value of a boolean

Increment and Decrement Operators

Java also provides increment and decrement operators: `++` and `--` respectively. `++` increases the value of the operand by **1**, while `--` decrease it by **1**. For example,

6. Java Bitwise Operators

Bitwise operators in Java are used to perform operations on individual bits. For example,

Here, `~` is a bitwise operator. It inverts the value of each bit (**0** to **1** and **1** to **0**).

The various bitwise operators present in Java are:

Operator	Description

~	Bitwise Complement
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift
&	Bitwise AND
^	Bitwise exclusive OR

These operators are not generally used in Java. To learn more, visit Java Bitwise and Bit Shift Operators.

Reading input using scanner class:

In Java, Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings.
Using the Scanner class in Java is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.

Java Scanner Input Types

Scanner class helps to take the standard input stream in Java. So, we need some methods to extract data from the stream. Methods used for extracting data are mentioned below:

Method	Description
nextBoolean()	Used for reading Boolean value
nextByte()	Used for reading Byte value
nextDouble()	Used for reading Double value
nextFloat()	Used for reading Float value
nextInt()	Used for reading Int value

Method	Description
nextLine()	Used for reading Line value
nextLong()	Used for reading Long value
nextShort()	Used for reading Short value

Let us look at the code snippet to read data of various data types.

```
Scanner sc = new Scanner(System.in);
int num = s.nextInt();
```

UNIT 2

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.

- ✓ Simple if statement
- ✓ if-else statement
- ✓ if-else-if ladder
- ✓ Nested if-statement

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:

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

EXAMPLE:

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

Output:

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

2) if-else statement

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

Syntax:

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

EXAMPLE:

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

```
} else {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

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

3) if-else-if ladder:

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

Syntax :

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

EXAMPLE:

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

Output:

```
Delhi
```

4. Nested if-statement

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

Syntax :

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

EXAMPLE:

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

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

Output:

```
Delhi
```

Switch Statement:

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

The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

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

syntax :

```
switch (expression){
  case value1:
   statement1;
   break;
  .
  .
  .
  case valueN:
   statementN;
   break;
  default:
   default statement;
}
```

EXAMPLE:.

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

```
}
}
```

Output:

```
2
```

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

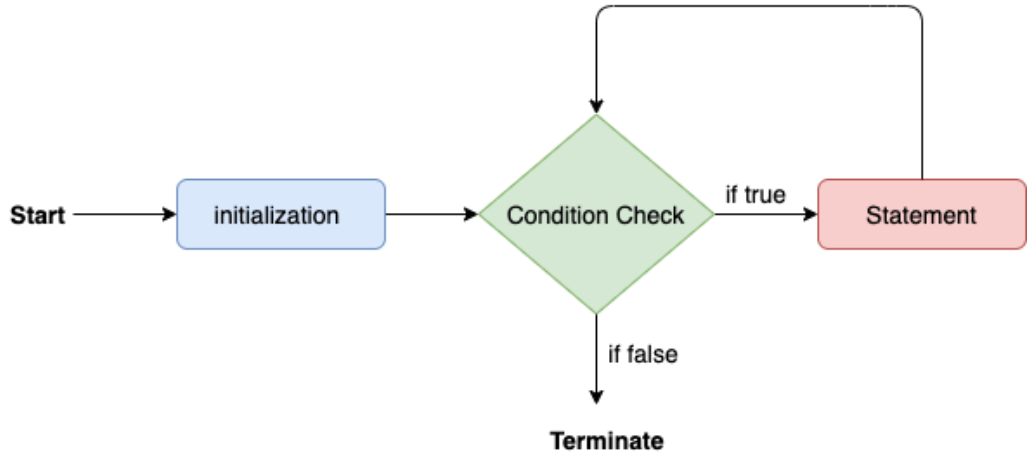
Let's understand the loop statements one by one.

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.



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

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. }

EXAMPLE:

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

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

Java while loop

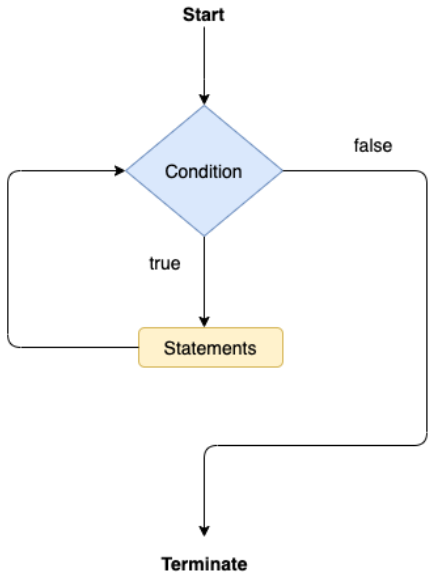
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while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

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

The syntax of the while loop is given below.

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

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



EXAMPLE:

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

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

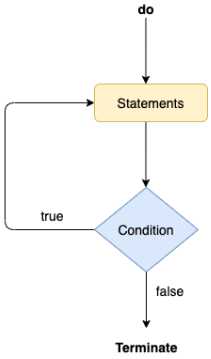
Java do-while loop

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

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

1. **do**
2. {
3. //statements
4. } **while** (condition);

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



EXAMPLE:

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

Output:

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

Jump Statements:

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

Java break statement

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

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

The break statement example with for loop

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

BreakExample.java

```
public class BreakExample {

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

Output:

```
0
1
2
3
4
```

```
5
6
```

break statement example with labeled for loop

Calculation.java

```
public class Calculation {

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

Output:

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

Java continue statement

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

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

```
public class ContinueExample {

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

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

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

if(j == 4) {
```

```
    continue;
    }
    System.out.println(j);
    } }}}
```

Output:

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

Java Arrays:

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

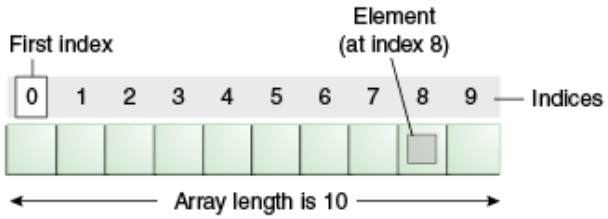
Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

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.

//Java Program to illustrate how to declare, instantiate, initialize

```
//and traverse the Java array.
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
10
20
70
40
50
```

Example of Multidimensional Java Array

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

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
  System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
```

```
}}
```

Output:

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

Java String Class Methods

The **java.lang.String** class provides a lot of built-in methods that are used to manipulate **string in Java**. By the help of these methods, we can perform operations on String objects such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a String if you submit any form in window based, web based or mobile application.

Let's use some important methods of String class.

Java String toUpperCase() and toLowerCase() method

The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.

Stringoperation1.java

```
public class Stringoperation1
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
}
}
```

Output:

```
SACHIN
sachin
Sachin
```

Java String trim() method

The String class trim() method eliminates white spaces before and after the String.

Stringoperation2.java

```
public class Stringoperation2
{
public static void main(String ar[])
{
String s="  Sachin  ";
System.out.println(s);//  Sachin
System.out.println(s.trim());//Sachin
}
}
```

Output:

```
Sachin
Sachin
```

Java String startsWith() and endsWith() method

The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

Stringoperation3.java

```
public class Stringoperation3
{
public static void main(String ar[])
{
String s="Sachin";
 System.out.println(s.startsWith("Sa"));//true
 System.out.println(s.endsWith("n"));//true
}
}
```

Output:

```
true
true
```

Java String charAt() Method

The String class charAt() method returns a character at specified index.

Stringoperation4.java

```
public class Stringoperation4
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
}
}
```

Output:

```
S
h
```

Java String length() Method

The String class length() method returns length of the specified String.

Stringoperation5.java

```
public class Stringoperation5
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.length());//6
}
}
```

Output:

```
6
```

Java String intern() Method

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a String equal to this String object as determined by the equals(Object) method, then the String from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

Stringoperation6.java

```
public class Stringoperation6
{
public static void main(String ar[])
{
String s=new String("Sachin");
String s2=s.intern();
System.out.println(s2);//Sachin
}
}
```

Output:

sachin

UNIT 3

Classes and Objects:

Create Object in Java

The **object** is a basic building block of an OOPs language. In **Java**, we cannot execute any program without creating an **object**. There is various way to **create an object in Java** that we will discuss in this section, and also learn **how to create an object in Java.**

Using new Keyword

Using the **new** keyword is the most popular way to create an object or instance of the class. When we create an instance of the class by using the new keyword, it allocates memory (heap) for the newly created **object** and also returns the **reference** of that object to that memory. The new keyword is also used to create an array. The syntax for creating an object is:

1. ClassName object = **new** ClassName();

Let's create a program that uses new keyword to create an object.

CreateObjectExample1.java

```
public class CreateObjectExample1
{
void show()
{
System.out.println("Welcome to javaTpoint");
}
public static void main(String[] args)
{
//creating an object using new keyword
CreateObjectExample1 obj = new CreateObjectExample1();
//invoking method using the object
obj.show();
}  }
```

Output:

```
Welcome to javaTpoint
```

Instance Variable in Java

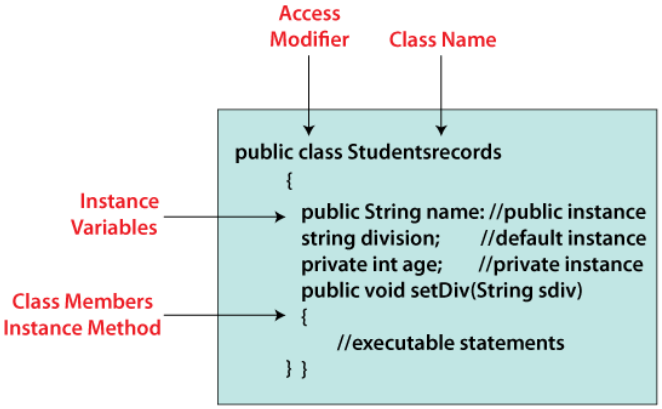
In any programming language, the program needs identifiers for storing different values that can be used throughout the program. These identifiers are variables.

Variable in Java

- A variable is a name assigned to a value that is stored inside the system memory. The value can be updated during the program execution.
- In Java programming, the variables used for the program need to declare them first.
- The variable is declared using a data type followed by the identifier name. The variable can be initialized at the time of declaration or it can be assigned a value taken from the user during the program execution.
- There are basically three types of variables in Java,
    1. Java Local variable
    2. Java Instance variable
    3. Java Static variable / Java class variable

Java Instance Variable

- The variables that are declared inside the class but outside the scope of any method are called instance variables in Java.
- The instance variable is initialized at the time of the class loading or when an object of the class is created.
- An instance variable can be declared using different access modifiers available in Java like default, private, public, and protected.
- Instance variables of different types have default values that are specified in the next point.



Features

1. To use an instance variable an object of the class must be created.
2. An instance variable is destroyed when the object it is associated with is destroyed.
3. An instance variable does not compulsory need to be initialized.
4. Instance variables are accessible inside the same class that declares them.

Limitations of Instance Variable

1. It cannot be declared static, abstract, striftp, synchronized, and native.
2. It can be declared final and transient.
3. It can be of any of the four access modifiers available in Java (private, public, protected, and default).

Default Values of Instance Variables in Java

The instance variables in Java are of different data types as follows;

Instance variable type	Default values
boolean	false
byte	(byte) 0
short	(short) 0
int	0
double	0.0d
float	0.0
long	0L
Object	null
char	\u0000

Let's use instance variable in Java program.

Access Modifiers in Java

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

Constructors in Java

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

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

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>(){}

2.

Parameterized Constructor

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

Method Overloading in Java

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

---

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. }}
10. 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
```

Static Members

In Java, static members are those which belongs to the class and you can access these members without instantiating the class.

The static keyword can be used with methods, fields, classes (inner/nested), blocks.

Static Methods – You can create a static method by using the keyword *static*. Static methods can access only static fields, methods. To access static methods there is no need to instantiate the class, you can do it just using the class name as –

Example

Live Demo

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

Output

Hello

Static Fields – You can create a static field by using the keyword static. The static fields have the same value in all the instances of the class. These are created and initialized when the class is loaded for the first time. Just like static methods you can access static fields using the class name (without instantiation).

Example

Live Demo

```
public class MyClass {
  public static int data = 20;
  public static void main(String args[]){
    System.out.println(MyClass.data);
  }
```

```
  Java Arrays with Answers
  27
}
```

Output

20

Static Blocks – These are a block of codes with a static keyword. In general, these are used to initialize the static members. JVM executes static blocks before the main method at the time of class loading.

Example

Live Demo

```
public class MyClass {
  static{
    System.out.println("Hello this is a static block");
  }
  public static void main(String args[]){
    System.out.println("This is main method");
  }
}
```

Output

Hello this is a static block

This is main method

Inheritance

Inheritance is a mechanism of driving a new class from an existing class. The existing (old) class is known as **base class** or **super class** or **parent class**. The new class is known as a **derived class** or **sub class** or **child class**. It allows us to use the properties and behavior of one class (parent) in another class (child).

A class whose properties are inherited is known as **parent class** and a class that inherits the properties of the parent class is known as **child class**. Thus, it establishes a relationship between parent and child class that is known as parent-child or **Is-a** relationship.

Suppose, there are two classes named **Father** and **Child** and we want to inherit the properties of the Father class in the Child class. We can achieve this by using the **extends** keyword.

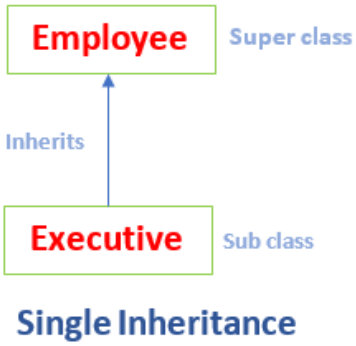
Types of Inheritance

Java supports the following four types of inheritance:

- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes it is also known as **simple inheritance**.

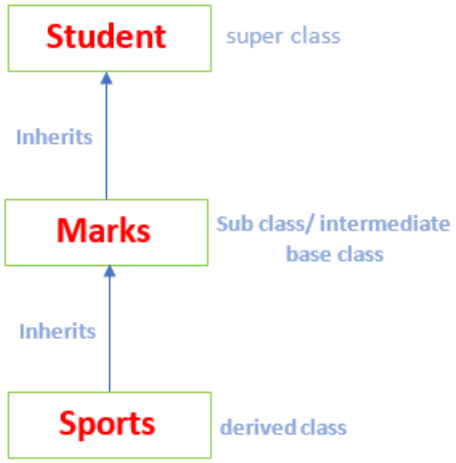


Single Inheritance

In the above figure, Employee is a parent class and Executive is a child class. The Executive class inherits all the properties of the Employee class.

Multi-level Inheritance

In **multi-level inheritance**, a class is derived from a class which is also derived from another class is called multi-level inheritance. In simple words, we can say that a class that has more than one parent class is called multi-level inheritance. Note that the classes must be at different levels. Hence, there exists a single base class and single derived class but multiple intermediate base classes.

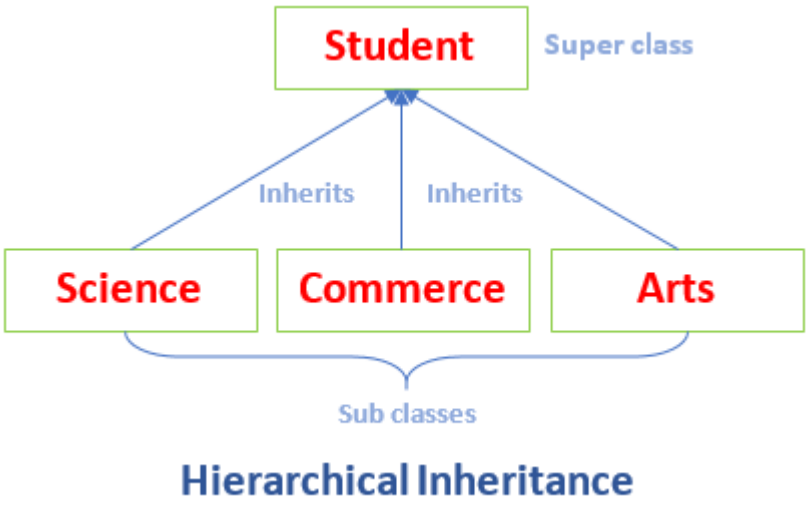


Multi-level Inheritance

In the above figure, the class Marks inherits the members or methods of the class Students. The class Sports inherits the members of the class Marks. Therefore, the Student class is the parent class of the class Marks and the class Marks is the parent of the class Sports. Hence, the class Sports implicitly inherits the properties of the Student along with the class Marks.

Hierarchical Inheritance

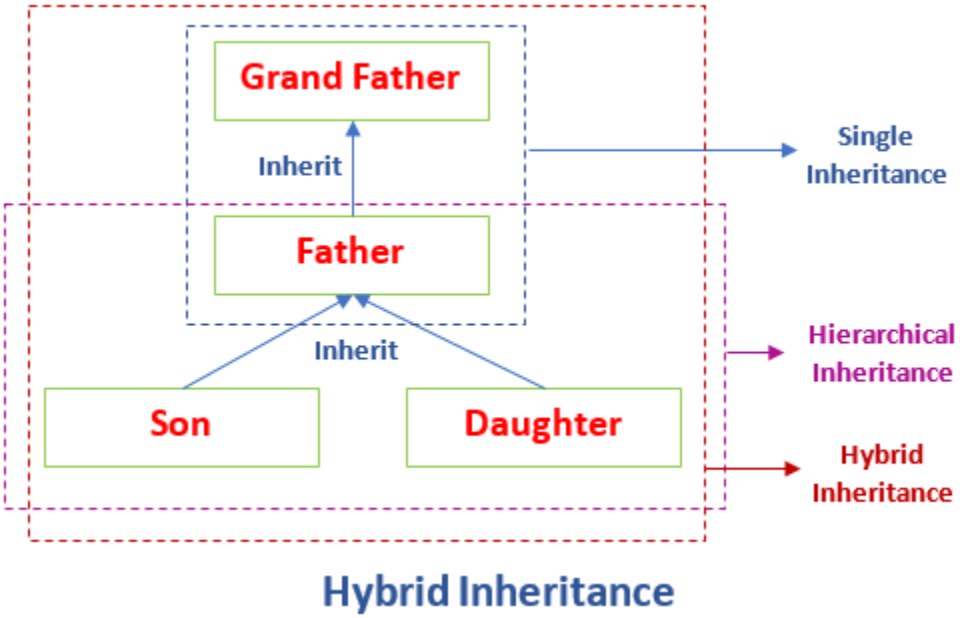
If a number of classes are derived from a single base class, it is called **hierarchical inheritance**.



In the above figure, the classes Science, Commerce, and Arts inherit a single parent class named Student.

Hybrid Inheritance

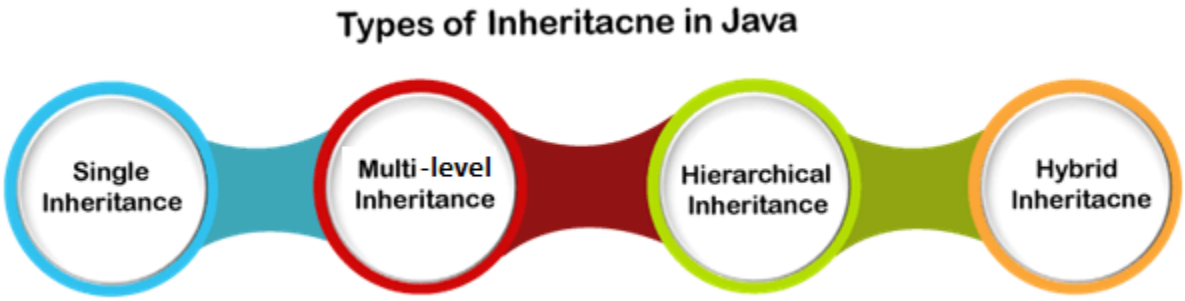
Hybrid means consist of more than one. Hybrid inheritance is the combination of two or more types of inheritance.



In the above figure, GrandFather is a super class. The Father class inherits the properties of the GrandFather class. Since Father and GrandFather represents single inheritance. Further, the Father class is inherited by the Son and Daughter class. Thus, the Father becomes the parent class for Son and Daughter. These classes represent the hierarchical inheritance. Combinedly, it denotes the hybrid inheritance.

Multiple Inheritance (not supported)

Java does not support multiple inheritances due to ambiguity. For example, consider the following Java program.



Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1. //Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
4. //Creating a parent class
5. class Vehicle{
6.   void run(){System.out.println("Vehicle is running");}
7. }
8. //Creating a child class
9. class Bike extends Vehicle{
10.  public static void main(String args[]){
11.  //creating an instance of child class
12.  Bike obj = new Bike();
13.  //calling the method with child class instance
14.  obj.run();
15.  }
16. }
```

Output:

```
Vehicle is running
```

Final Keyword In Java

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

- ✓ variable
- ✓ method
- ✓ 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).

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. }
```

Test it Now

Output:Compile Time Error

Abstract Method in Java

In object oriented programming, abstraction is defined as hiding the unnecessary details (implementation) from the user and to focus on essential details (functionality). It increases the efficiency and thus reduces complexity.

In Java, abstraction can be achieved using abstract classes and methods. In this tutorial, we will learn about abstract methods and its use in Java.

Abstract class

A class is declared abstract using the **abstract** keyword. It can have zero or more abstract and non-abstract methods. We need to extend the abstract class and implement its methods. It cannot be instantiated.

Syntax for abstract class:

1. **abstract class** class_name {
2. //abstract or non-abstract methods
3. }

Note: An abstract class may or may not contain abstract methods.

Abstract Method

A method declared using the **abstract** keyword within an abstract class and does not have a definition (implementation) is called an abstract method.

When we need just the method declaration in a super class, it can be achieved by declaring the methods as abstracts.

Abstract method is also called subclass responsibility as it doesn't have the implementation in the super class. Therefore a subclass must override it to provide the method definition.

Syntax for abstract method:

1. **abstract** return_type method_name([argument-list]);

Here, the abstract method doesn't have a method body. It may have zero or more arguments.

Points to Remember

Following points are the important rules for abstract method in Java:

- An abstract method do not have a body (implementation), they just have a method signature (declaration). The class which extends the abstract class implements the abstract methods.
- If a non-abstract (concrete) class extends an abstract class, then the class must implement all the abstract methods of that abstract class. If not the concrete class has to be declared as abstract as well.
- As the abstract methods just have the signature, it needs to have semicolon (;) at the end.
- Following are various **illegal combinations**of other modifiers for methods with respect to *abstract* modifier:
    - final
    - abstract native
    - abstract synchronized
    - abstract static

- abstract private
- abstract strictfp

- If a class contains abstract method it needs to be abstract and vice versa is not true.

Interface in Java

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.

Multiple Inheritance in Java

Introduction

The concept of inheritance, which enables classes to adopt features and attributes from other classes, is fundamental to object-oriented programming. Due to Java's support for single inheritance, a class can only descend from one superclass. However, Java offers a method for achieving multiple inheritances through interfaces, enabling a class to implement many interfaces. We will examine the idea of multiple inheritance in Java, how it is implemented using interfaces, and use examples to help us understand.

Understanding Multiple Inheritance A class's capacity to inherit traits from several classes is referred to as multiple inheritances. This notion may be quite helpful when a class needs features from many sources. Multiple inheritances, however, can result in issues like the diamond problem, which occurs when two superclasses share the same method or field and causes conflicts. Java uses interfaces to implement multiple inheritances in order to prevent these conflicts.

Java interfaces

A Java interface is a group of abstract methods that specify the behavior that implementing classes must follow. It serves as a class blueprint by outlining each class's methods. Interfaces offer a degree of abstraction for specifying behaviors but cannot be instantiated like classes. In Java, a class can successfully implement several interfaces to achieve multiple inheritance.

Syntax of implementing multiple interfaces:

To implement multiple interfaces in Java, the following syntax is used:

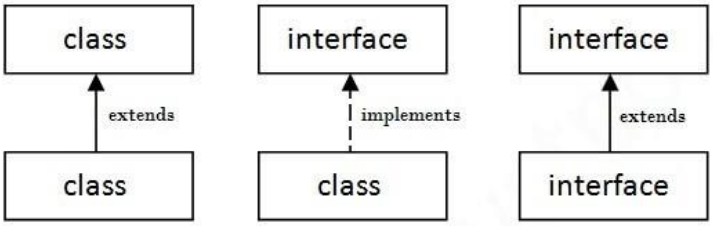
```
1. class MyClass implements Interface1, Interface2, Interface3 {
2.     // class body
```

3. }

The classes "MyClass" and "Interface1", "Interface2", and "Interface3" can now inherit and implement methods from other interfaces. As a result, the class is able to display the behaviours specified in every interface it implements.

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.

1. **interface** printable{
2. **void** print();
3. }
4. **class** A6 **implements** printable{
5. **public void** print(){System.out.println("Hello");}
6.
7. **public static void** main(String args[]){
8. A6 obj = **new** A6();
9. obj.print();
10. }
11. }

Output:

```
Hello
```

Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

File: TestInterface1.java

```
1. //Interface declaration: by first user
2. interface Drawable{
3. void draw();
4. }
5. //Implementation: by second user
6. class Rectangle implements Drawable{
7. public void draw(){System.out.println("drawing rectangle");}
8. }
9. class Circle implements Drawable{
10. public void draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. class TestInterface1{
14. public static void main(String args[]){
15. Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDr
    awable()
16. d.draw();
17. }}
```

Output:

```
drawing circle
```

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```
1. interface Bank{
2. float rateOfInterest();
```

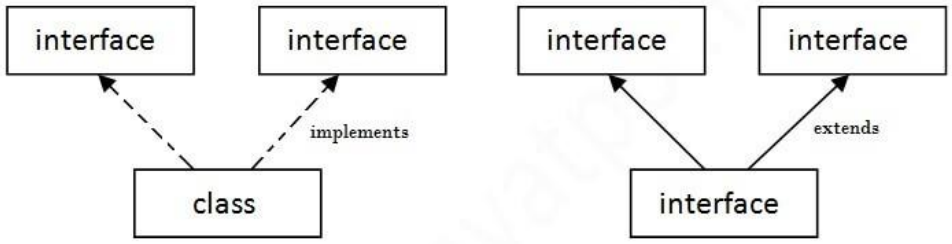
```
3. }
4. class SBI implements Bank{
5. public float rateOfInterest(){return 9.15f;}
6. }
7. class PNB implements Bank{
8. public float rateOfInterest(){return 9.7f;}
9. }
10. class TestInterface2{
11. public static void main(String[] args){
12. Bank b=new SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}
```

Output:

```
ROI: 9.15
```

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

UNIT 4

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.

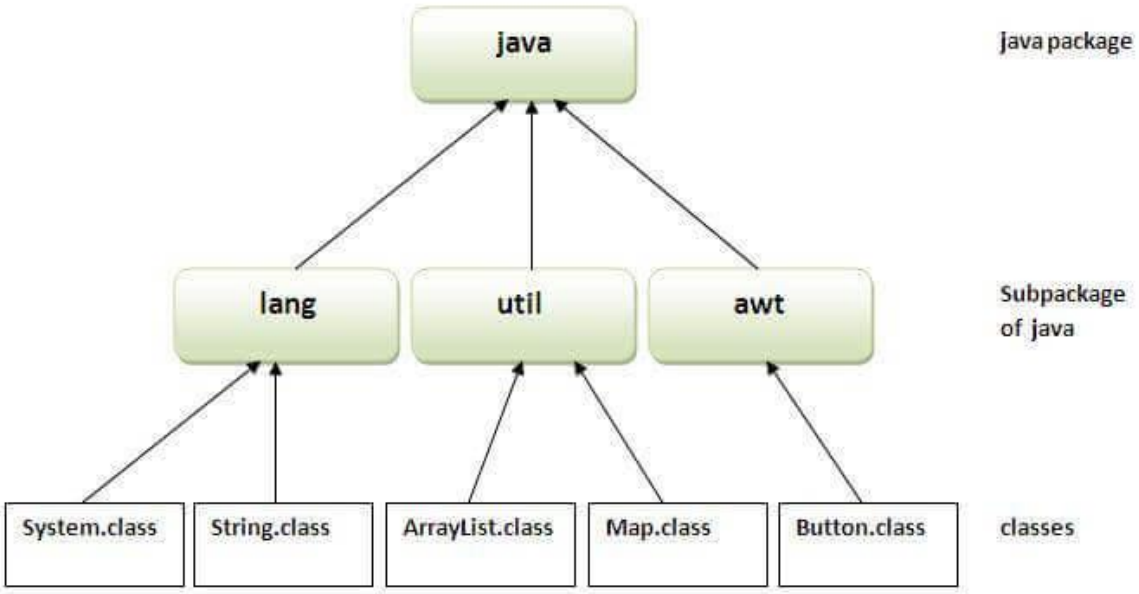
Here, we will have the detailed learning of creating and using user-defined packages.

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.

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.    System.out.println("Welcome to package");
6.   }
7. }
```

Types of Packages in Java

Packages in Java can be categorised into 2 categories.

1. Built-in / predefined packages
2. User-defined packages.

Let's understand them in a little more detail.

Built-in packages

When we install **Java** on a personal computer or laptop, many packages are automatically installed. Each of these packages is unique and capable of handling various tasks. This eliminates the need to build everything from scratch. Here are some examples of built-in packages in Java:

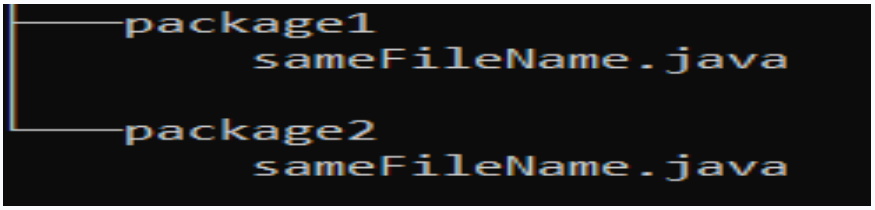
- **java.lang**
- **java.io**
- **java.util**
- **java.applet**
- **java.awt**
- **java.net**

User-defined packages

User-defined packages in Java are those that developers create to incorporate different needs of applications. In simple terms, User-defined packages are those that the users define. Inside a package, you can have Java files like classes, interfaces, and a package as well (called a sub-package).

Handling Name Conflicts

Having two classes with the same name in a single file produces a name collision error, but having them on different packages can prevent it. So, packages in Java also help in preventing naming collision errors. See example.



We've made two packages, package1 and package2. Both packages contain a file of the same name that is **sameFileName.java**. Having them inside a single package causes a name collision error, and we use two packages to thwart it.

Threads

A thread is a single sequence stream within a process. Threads are also called lightweight processes as they possess some of the properties of processes. Each thread belongs to exactly one process. In an operating system that supports multithreading, the process can consist of many threads.

Lifecycle and States of a Thread in Java

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

- ✓ New State
- ✓ Runnable State
- ✓ Blocked State
- ✓ Waiting State
- ✓ Timed Waiting State
- ✓ Terminated State

The diagram shown below represents various states of a thread at any instant in time.

Lifecycle-and-States-of-a-Thread-in-Java

Life Cycle of a Thread
There are multiple states of the thread in a lifecycle as mentioned below:

New Thread: When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.

Runnable State: A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

Blocked: The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.

Waiting state: The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.

Timed Waiting: A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
Terminated State: A thread terminates because of either of the following reasons:
Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

Exception Handling in Java

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.

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:

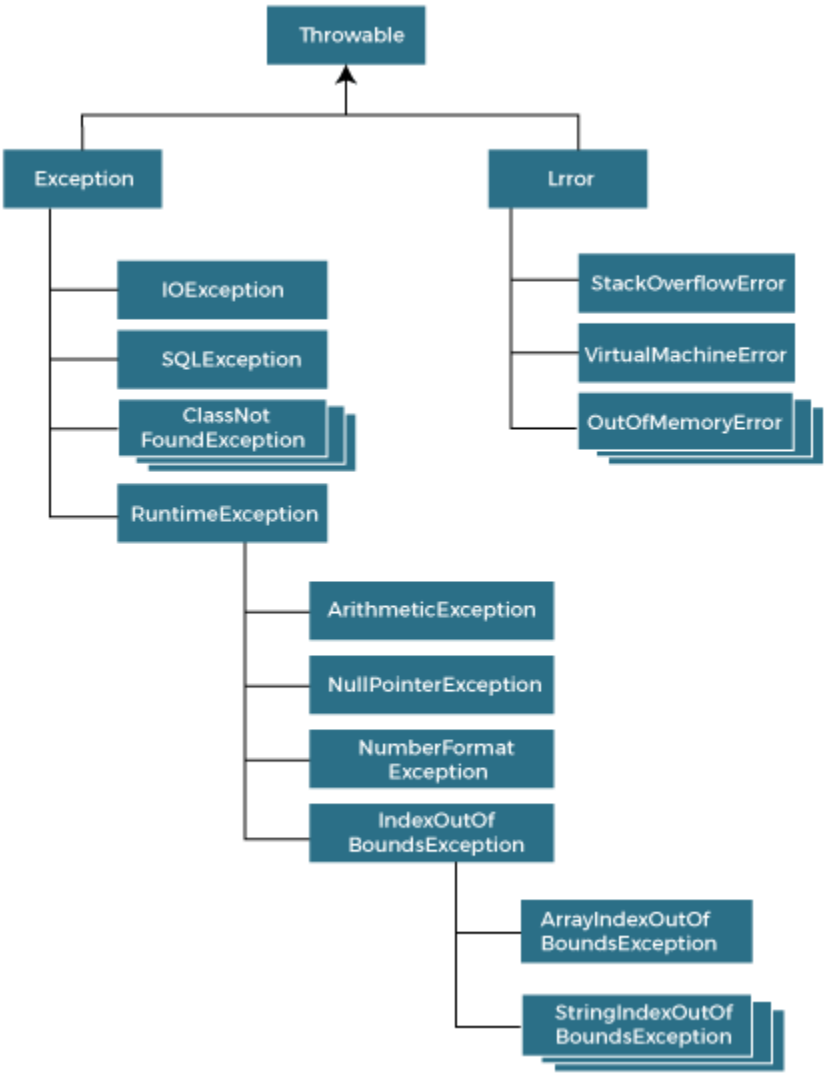
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

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.
Do You Know?

- What is the difference between checked and unchecked exceptions?
- What happens behind the code int data=50/0;?
- Why use multiple catch block?
- Is there any possibility when the finally block is not executed?
- What is exception propagation?
- What is the difference between the throw and throws keyword?
- What are the 4 rules for using exception handling with method overriding?

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.

Java Exception Keywords

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

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 Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

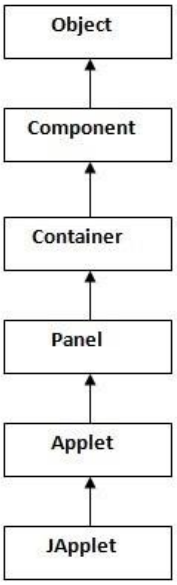
Drawback of Applet

- Plugin is required at client browser to execute applet.

Do You Know

- Who is responsible to manage the life cycle of an applet ?
- How to perform animation in applet ?
- How to paint like paint brush in applet ?
- How to display digital clock in applet ?
- How to display analog clock in applet ?
- How to communicate two applets ?

Hierarchy of Applet



As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.