LECTURE NOTES ON

Object Oriented Programming Using JAVA

Prepared by

Er. Jaykrushna Mohanta

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
GANESH INSTITUTE OF ENGINEERING & TECHNOLOGY
ANDHARUA,BHUBANESWAR

SYLLABUS

PRINCIPLES OF OBJECT ORIENTED PROGRAMMING
- Procedure Oriented paradigm
- Object oriented paradigm

DATA TYPES & I/O OPERATIONS
- Basic data types
- User defined data types & derived data types
- Dynamic initialization of variables
- Operators & expressions
- Formatted & unformatted I/O

CLASSES
- Introduction to classes.
- Member functions
- Static Data Member
- Arrays within a class
- Pointers to members

CONSTRUCTOR AND DESTRUCTOR
- The purpose of Constructor & Destructor.
- Constructors with parameters
- Multiple constructors in a class
- Dynamic initialization of objects
- Destructors

INHERITANCE OF CLASSES 10
- Derived classes
- Single inheritance
- Multilevel and Multiple inheritance
- Hierarchical inheritance
- Virtual Base Classes

POLYMORPHISM 04
- Fundamental idea on Polymorphism
- Pointer to objects & derived classes
- Virtual Functions

FILE HANDLING 07
- Streams and stream classes
- Classes for file stream operation
- Opening and closing files
- How to handle Error
- Command line arguments

TEMPLATES AND EXCEPTION HANDLING 04
- Class templates & Function Templates
- Template Arguments
- Exception Handling

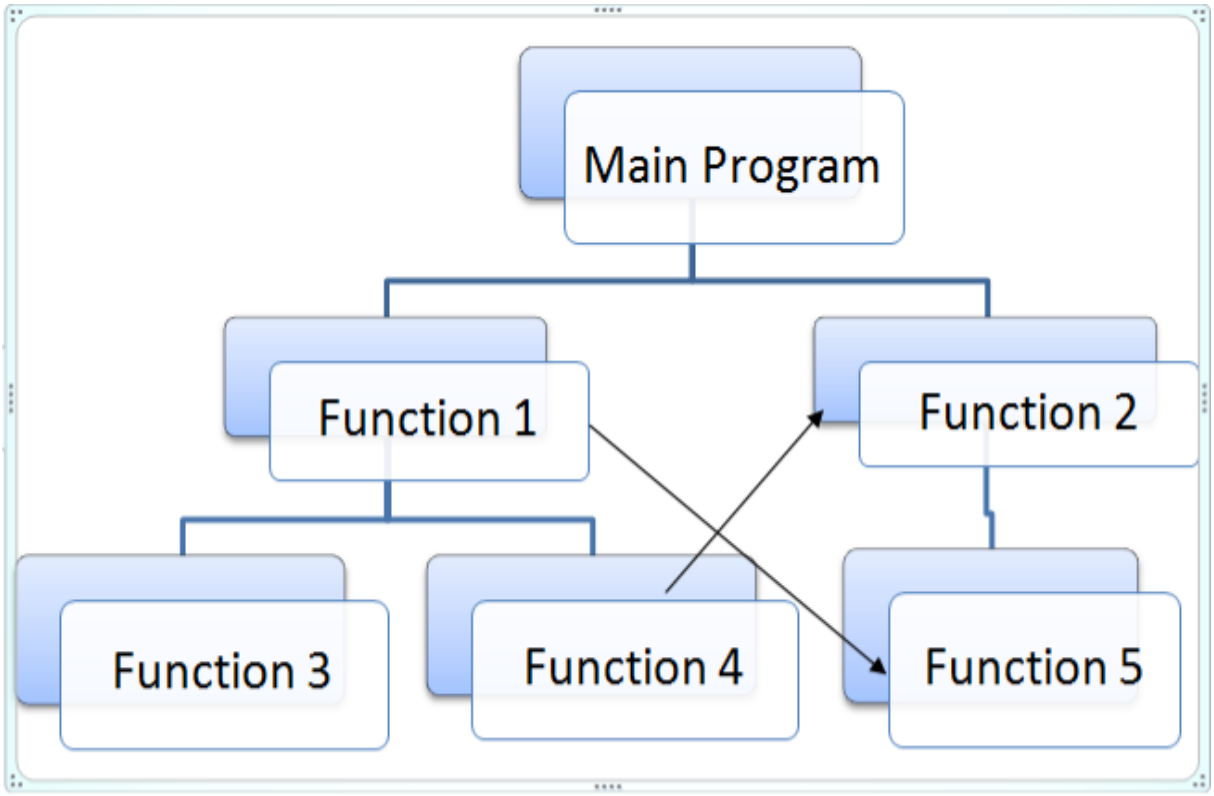
CHAPTER-1

PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING

Procedure Oriented paradigm

Procedure oriented programming is a set of functions. In this program C language is used. To perform any particular task, set of function are compulsory. For example, a program may involve collecting data from user, performing some kind of calculation on that data and printing the data onscreen when is requested.

POP method also emphases the functions or the subroutines.



Object Oriented paradigm:

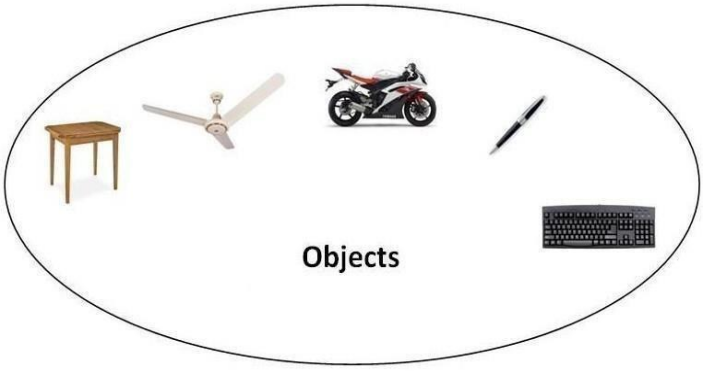
Object Oriented Programming is a paradigm that provides many concepts such as inheritance, data binding, polymorphism etc.

Simula is considered as the first object-oriented programming language. The programmingparadigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

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

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



Apart from these concepts, there are some other terms which are used in Object Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition

Object

Any entity that has state and behaviour 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.

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.

Advantage of OOPs over Procedure-oriented programming language

1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.

2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

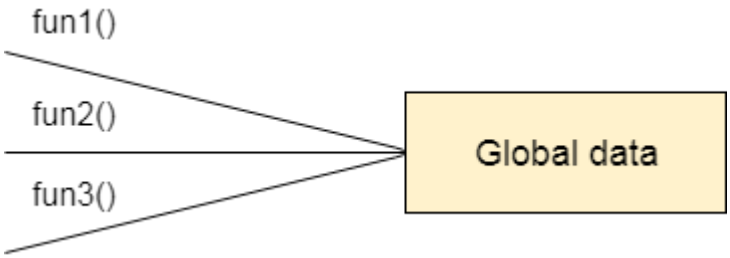


Figure: Data Representation in Procedure-Oriented Programming

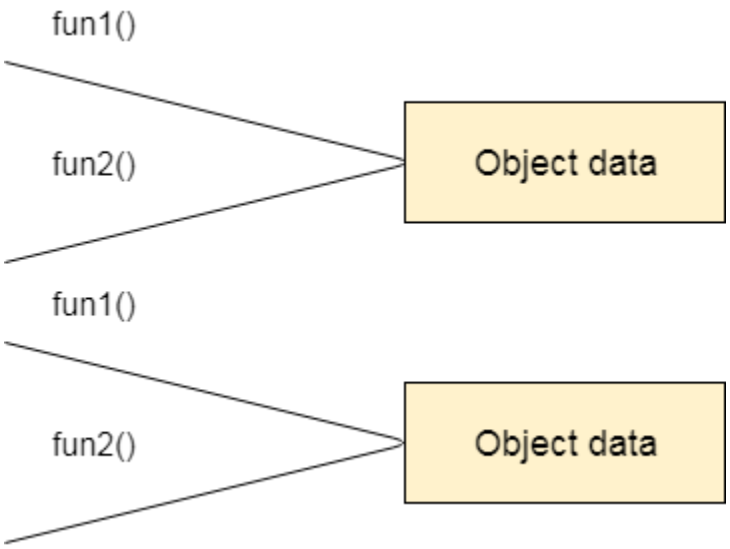


Figure: Data Representation in Object-Oriented Programming

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

What is the difference between an object-oriented programming language and object-based programming language?

Object-based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object-based programming languages.

Procedural and object-oriented programming paradigms

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

History of Java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

1) **James Gosling, Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**
2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.
3) Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.
4) After that, it was called **Oak** and was developed as a part of the Green project.
5) Why Oak? Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.
6) In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.
7) Why had they chose the name Java for Java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say. According to James Gosling, "Java was one of the top choices along with Silk". Since Java was so unique, most of the team members preferred Java than other names.
8) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.
9) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.
10)

Java Version History

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)
4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th Mar 2014)
11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th Mar 2018)
13. Java SE 11 (September 2018)
14. Java SE 12 (March 2019)
15. Java SE 13 (September 2019)
16. Java SE 14 (Mar 2020)
17. Java SE 15 (September 2020)
18. Java SE 16 (Mar 2021)
19. Java SE 17 (September 2021)
20. Java SE 18 (to be released by March 2022)

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

Java Comments

The Java comments are the statements in a program that are not executed by the compiler and interpreter.

Types of Java Comments

There are three types of comments in Java.

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

1) Java Single Line Comment

The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements. Single line comments start with two forward slashes **(//)**. Any text in front of // is not **Syntax:**

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

Let's use single line comment in a Java program.

CommentExample1.java

```
1. public class CommentExample1 {
2. public static void main(String[] args) {
3. int i=10; // i is a variable with value 10
4. System.out.println(i); //printing the variable i
5. }
6. }
```

Output:

```
10
```

2) Java Multi Line Comment

The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there).

Multi-line comments are placed between /* and */. Any text between /* and */ is not executed by Java.

Syntax:

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

CommentExample2.java

```
1.  public class CommentExample2 {
2.  public static void main(String[] args) {
3.  /* Let's declare and
4.   print variable in java. */
5.    int i=10;
6.       System.out.println(i);
7.  /* float j = 5.9;
8.       float k = 4.4;
9.       System.out.println( j + k ); */
10. }
11. }
```

Output:

```
10
```

3) Java Documentation Comment

Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API. These APIs are needed for reference, i.e., which classes, methods, arguments, etc., are used in the code.

To create documentation API, we need to use the **Javadoc tool**. The documentation comments are placed between /** and */.

Syntax:

```
1. /**
2. *
3.  *We can use various tags to depict the parameter
4.  *or heading or author name
5.  *We can also use HTML tags
6. *
7. */
```

CHAPTER-2

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.

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:

```
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:

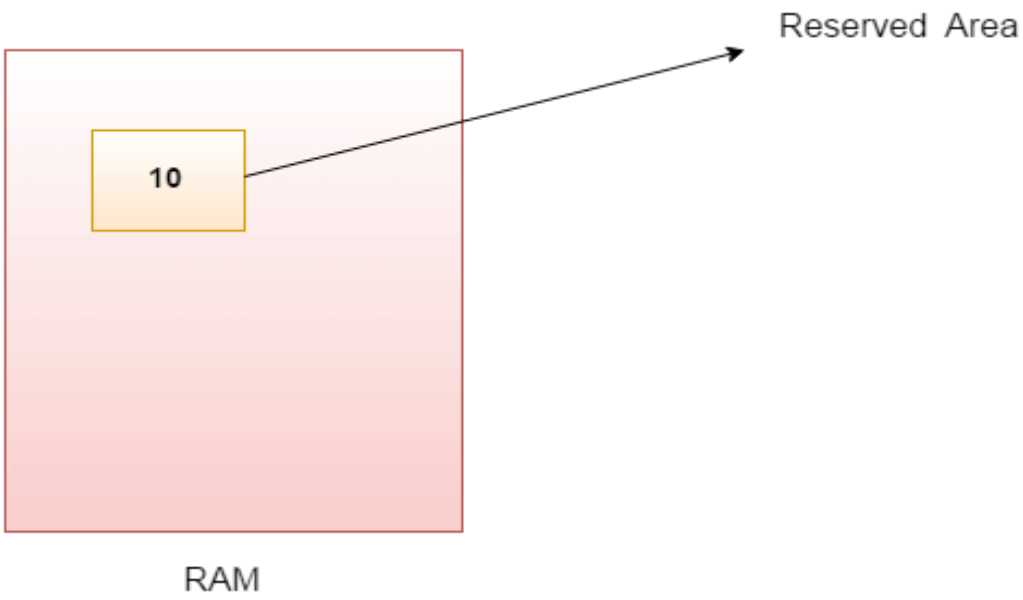
```
1. char letterA = 'A'
```

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.




```
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
```

Java Variable Example: Add Two Numbers

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

Output:

```
20
```

Java Variable Example: Widening

```
1.  public class Simple{
2.  public static void main(String[] args){
3.  int a=10;
4.  float f=a;
5.  System.out.println(a);
6.  System.out.println(f);
7. }
8.  }
```

Output:

```
10
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. }}
```

Java Constant

As the name suggests, a **constant** is an entity in programming that is immutable. In other words, the value that cannot be changed. In this section, we will learn about **Java constant** and **how to declare a constant in Java**.

What is constant?

Constant is a value that cannot be changed after assigning it. Java does not directly support the constants. There is an alternative way to define the constants in Java by using the non-access modifiers static and final.

Properties	Primitive data types	Objects
Origin	Pre-defined data types	User-defined data types
Stored structure	Stored in a stack	Reference variable is stored in stack and the original object is stored in heap
When copied	Two different variables are created along with different assignment (only values are same)	Two reference variable is created but both are pointing to the same object on the heap
When changes are made in the copied variable	Change does not reflect in the original ones.	Changes reflected in the original ones.
Default value	Primitive datatypes do not have null as default value	The default value for the reference variable is null
Example	byte, short, int, long, float, double, char, Boolean	array, string class, interface etc.

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	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Expression in Java

An expression in Java is a series of operators, variables, and method calls constructed according to the syntax given, the language for evaluating a single value is as follows: For instance,

int marks;

marks = 90;

Here marks=90 is an expression that returns an int value. Example:

```
Double a = 2.2, b = 3.4, result;
result = a + b - 3.4;
```

Here in this example,a+b-3.4 is an expression.

Simple Expressions

A simple expression is a literal method call or variable name without any usage of an operator. For example:

```
43                  // integer literal
name                  // variable name
System.out.println("Hello"); // method call
"Java"                // string literal
133B                  // double precision floating-point literal
32L                   // long integer literal
```

A simple expression in java has a type that can either be a primitive type or a reference type. In this example, 43 is a 32-bit integer, java is a string, 32L is a long 64-bit integer, etc.

Compound Expressions

It generally involves the usage of operators. It comprises one or more simple expressions, which are then integrated into a larger expression by using the operator. Consider another example in order to understand more clearly.

```
Double a = 2.3, b = 3.2, number;
number = a + b - 3.4;
```

ENUM

The **Enum in Java** is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

Enums are used to create our own data type like classes. The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more *powerful*. Here, we can define an enum either inside the class or outside the class.

Simple Example of Java Enum

```
1. class EnumExample1{
2. //defining the enum inside the class
3. public enum Season { WINTER, SPRING, SUMMER, FALL }
4. //main method
5. public static void main(String[] args) {
6. //traversing the enum
7. for (Season s : Season.values())
8. System.out.println(s);
9. }}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
```

C language provide us console input/output functions. As the name says, the console input/output functions allow us to -

- Read the input from the keyboard by the user accessing the console.
- Display the output to the user at the console.

Note: These input and output values could be of any primitive data type. There are two kinds of console input/output functions:

There are two kinds of console input/output functions:

No.	Functions
1	***Formatted input/output functions.***
2	***Unformatted input/output functions.***

Formatted input/output functions:

Formatted console input/output functions are used to take a single or multiple inputs from the user at console and it also allows us to display one or multiple values in the output to the user at the console. For more on formatted input/output functions, please read formatted input functions.

- ***Unformatted input/output functions***

  Unformatted console input/output functions are used to read a ***single*** input from the user at console and italso allows us to display the value in the output to the user at the console.

Some of the most important formatted console input/output functions are –

Functions	Description
getch()	Reads a *single* character from the user at the console, ***without*** *echoing it*.
getche()	Reads a *single* character from the user at the console, *and echoing it*.
getchar()	Reads a *single* character from the user at the console, *and*
gets()	Reads a single string entered by the user at the console.

puts()	Displays a single string's value at the console.
putch()	Displays a single character value at the console.
putchar()	Displays a single character value at the console.

CHAPTER-3

Classes in Java

A Java class is a set of object which shares common characteristics/ behaviour and common properties/ attributes. There are certain points about Java Classes as mentioned below:

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.

A Class in Java can contain:

- Data member
- Method
- Constructor
- Nested Class
- Interface

Declare Class in Java

```
access_modifier class<class_name>
{
   data member;
   method;
   constructor;
   nested class;
   interface;
}
```

Example:

- Animal
- Student
- Bird
- Vehicle
- Company

```
// Java Program for class example

class Student {
    int id;
    // data member (also instance variable)
```

```
    String name;

    public static void main (String args[])
    {
        // creating an object of
        // Student
        Student s1 = new Student();
        System.out.println(s1.id);
        System.out.println(s1.name);
    }
}
```

Components of Java Classes

In general, class declarations can include these components, in order:

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

Objects in Java

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

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

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*.

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
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

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

MEMBER FUNCTION

Member functions: The functions declared inside the class are known as member functions. Member functions are methods or functions that are defined inside of objects. Generally used to manipulate data members and other object data.

Method in Java

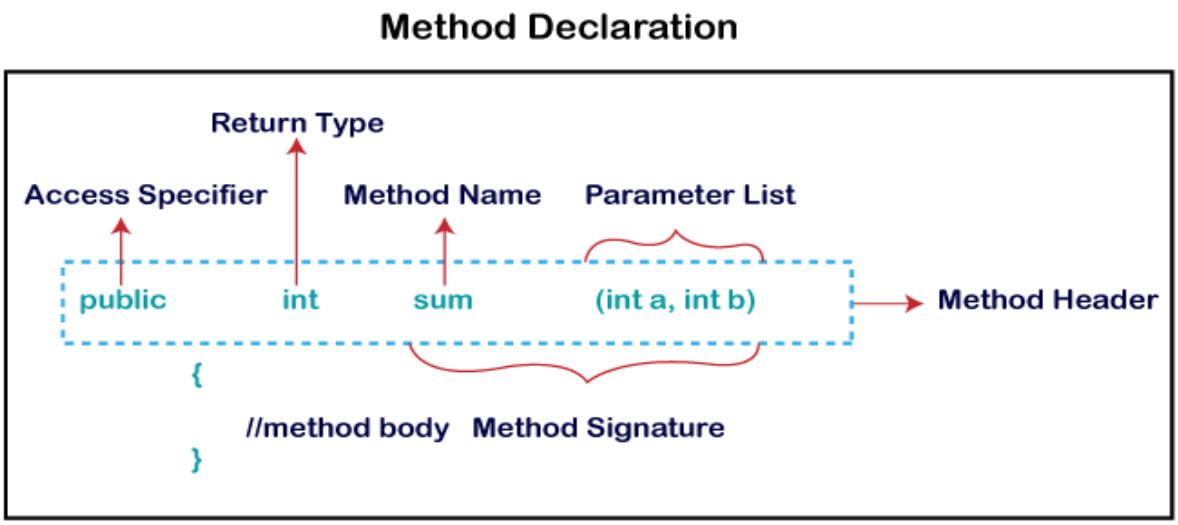
In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**. In this section, we will learn **what is a method in Java, types of methods, method declaration,** and **how to call a method in Java**.

What is a method in Java?

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.

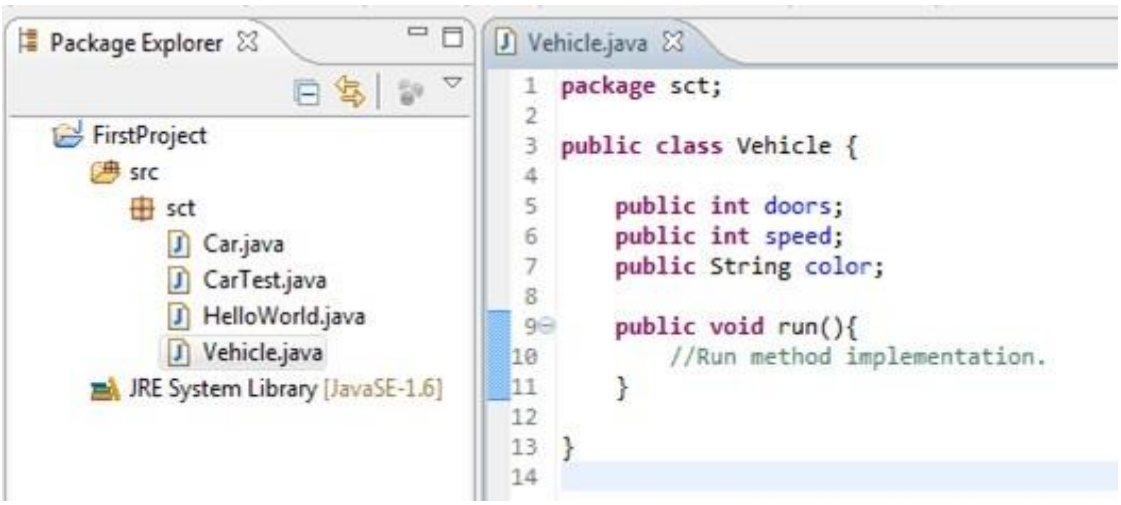


Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

public: Members (variables, methods, and constructors) declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in anotherpackage. Below screen shot shows eclipse view of public class with public members.



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

Example of public access modifier

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5. public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.  }
11. }
```

protected: The protected fields or methods, cannot be used for classes and Interfaces. Fields, methods andconstructors declared protected in a super-class can be accessed only by subclasses in other packages.

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

```
1. //save by A.java
2. package pack;
3. public class A{
4. protected void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4.
5. class B extends A{
6.   public static void main (String args[]){
```

```
7.      B obj = new B();
8.      obj.msg ();
9.     }
10. }
```

Output: Hello

Default

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

Example of default access modifier

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

```
1.  //save by A.java
2.  package pack;
3.  class A{
4.    void msg(){System.out.println("Hello");}
5.  }
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.  class B{
5.    public static void main(String args[]){
6.     A obj = new A();//Compile Time Error
7.    obj.msg();//Compile Time Error
8.    }
9.  }
```

private: The private (most restrictive) modifiers can be used for members but cannot be used for classes andInterfaces. Fields, methods or constructors declared

private are strictly controlled, which means they cannot be accessed by anywhere outside the enclosing class.

Simple example of private access modifier

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

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }
5.
6. public class Simple{
7.  public static void main(String args[]){
8.    A obj=new A();
9.    System.out.println(obj.data);//Compile Time Error
10.   obj.msg();//Compile Time Error
11.   }
12. }
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
1. class A{
2. private A(){}//private constructor
3. void msg(){System.out.println("Hello java");}
4. }
5. public class Simple{
6.  public static void main(String args[]){
7.    A obj=new A();//Compile Time Error
8.   }
9. }
```

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

TATIC DATA MEMBER

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 onlystatic fields, methods. To access static methods there is no need to instantiate the class, you can do it just usingthe class name as −

Example

Example 1: The static method does not have access to the instance variable

The JVM runs the static method first, followed by the creation of class instances. Because no objects are accessible when the static method is used. A static method does not have access to instance variables. As a result, a static method can't access a class's instance variable.

```
// Java program to demonstrate that
// The static method does not have
// access to the instance variable

import java.io.*;

public class GFG {
    // static variable
    static int a = 40;
```

```
// instance variable
int b = 50;

void simpleDisplay()
{
   System.out.println(a);
   System.out.println(b);
}

// Declaration of a static method.
static void staticDisplay()
{
  System.out.println(a);
}

// main method
public static void main(String[] args)
{
   GFG obj = new GFG();
   obj.simpleDisplay();

   // Calling static method.
   staticDisplay();
}}
```

ontrol 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

ecision-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

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.   }
.   }
```

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. }
```

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

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. }
```

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. }
```

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.

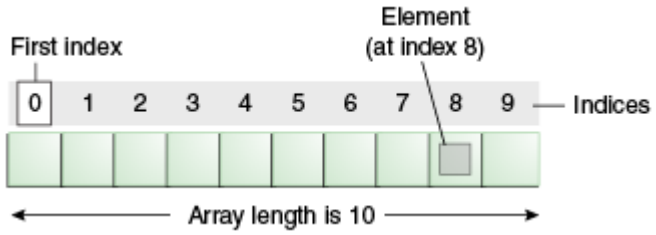
```
1. while(condition){
2. //looping statements
3. }
```

Java Arrays

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.

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 dimensional or multidimensional 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
```

Java Console Class

The Java Console class is be used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

```
1. String text=System.console().readLine();
2. System.out.println("Text is: "+text);
```

Java Console Example

```
1. import java.io.Console;
2. class ReadStringTest{
3. public static void main(String args[]){
4. Console c=System.console();
5. System.out.println("Enter your name: ");
6. String n=c.readLine();
7. System.out.println("Welcome "+n);
8. }
9. }
10. Output
11. Enter your name: Nakul Jain
12. Welcome Nakul Jain
```

Chapter-4

CONSTRUCTOR AND DESTRUCTOR 07

The purpose of Constructor & Destructor

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.

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>(){}
```

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 creatio

```
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. }
```

Output:

Bike is created

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. }
```

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. }
```

Output:

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

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. }
24. Output:
25. 111 Karan
26. 111 Karan
```

Java Destructor

In Java, when we create an object of the class it occupies some space in the memory (heap). If we do not delete these objects, it remains in the memory and occupies unnecessary space that is not upright from the aspect of programming. To resolve this problem, we use the **destructor**. In this section, we will discuss the alternate option to the **destructor in Java**. Also, we will also learn how to use the **finalize()** method as a destructor. The **destructor** is the opposite of the constructor. The constructor is used to initialize objects while the destructor is used to delete or destroy the object that releases the resource occupied by the object.

Remember that **there is no concept of destructor in Java**. In place of the destructor, Java provides the garbage collector that works the same as the destructor. The garbage collector is a program (thread) that runs on the JVM. It automatically deletes the unused objects (objects that are no longer used) and free-up the memory. The programmer has no need to manage memory, manually. It can be error-prone, vulnerable, and may lead to a memory leak.

Advantages of Destructor

- It releases the resources occupied by the object.
- No explicit call is required, it is automatically invoked at the end of the program execution.
- It does not accept any parameter and cannot be overloaded.

Example of Destructor

DestructorExample.java

```
1. public class DestructorExample
2. {
3. public static void main(String[] args)
4. {
5.  DestructorExample de = new DestructorExample ();
6.  de.finalize();
7.  de = null;
8.  System.gc();
9.  System.out.println("Inside the main() method");
10. }
11. protected void finalize()
12. {
13. System.out.println("Object is destroyed by the Garbage Collector");
14. }
15. }
```

Output

```
Object is destroyed by the Garbage Collector
Inside the main() method
Object is destroyed by the Garbage Collector
```

CHAPTER-5

OPERATOR OVERLOADING 07

Method in Java

In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**. In this section, we will learn **what is a method in Java, types of methods, method declaration,** and **how to call a method in Java**.

Creating Method

Considering the following example to explain the syntax of a method −

Syntax

Here,

- public static − modifier
- int − return type
- methodName − name of the method
- a, b − formal parameters
- int a, int b − list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax −

```
modifier returnType nameOfMethod (Parameter List) {
// method body
}
```

The syntax shown above includes −

- modifier − It defines the access type of the method and it is optional to use.
- returnType − Method may return a value.

- nameOfMethod − This is the method name. The method signature consists of the method name and the parameter list.

- Parameter List − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.

- method body − The method body defines what the method does with the statements.

Call by Value and Call by Reference in Java

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as cal being done in the called method, is not affected in the calling method.

Example of call by value in java

In case of call by value original value is not changed. Let's take a simple example:

```
1. class Operation{
2.  int data=50;
3.
4.  void change(int data){
5.  data=data+100;//changes will be in the local variable only
6.  }
7.
8.  public static void main(String args[]){
9.    Operation op=new Operation();
10.
11.   System.out.println("before change "+op.data);
12.   op.change(500);
13.   System.out.println("after change "+op.data);
14.
15. }
16. }
```

Output: before change 50
after change 50

Call By Reference

Java uses only call by value while passing reference variables as well. It creates a copy of references and passes them as valuable to the methods. As reference points to same address of object, creating a copy of reference is of no harm. But if new object is assigned to reference it will not be reflected.

Static Fields and Methods

Java static keyword

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. }
```

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
```

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. }
```

Output:
111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT

Java static block

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

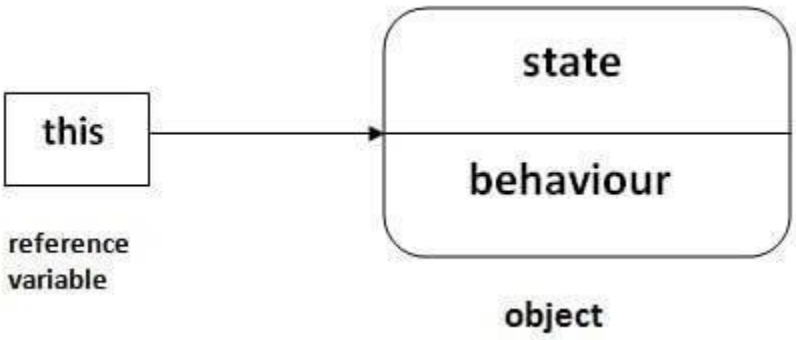
Example of static block

```
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

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.

```
7. class Student{
8. int rollno;
9. String name;
10. float fee;
11. Student(int rollno,String name,float fee){
12. this.rollno=rollno;
13. this.name=name;
14. this.fee=fee;
15. }
16. void display(){System.out.println(rollno+" "+name+" "+fee);}
17. }
18.
19. class TestThis2{
20. public static void main(String args[]){
21. Student s1=new Student(111,"ankit",5000f);
22. Student s2=new Student(112,"sumit",6000f);
23. s1.display();
24. s2.display();
25. }}
```

Output:

```
26. 111 ankit 5000.0
27. 112 sumit 6000.0
```

Difference between java Constructor and Method

Sr. No.	Constructor	Method

1.	A block of code that initialize at the time of creating a new object of the class is called constructor.	A set of statements that performs specific task with and without returning value to the caller is known as method.
2.	It is mainly used for initializing the object.	It is mainly used to reuse the code without writing the code again.
3.	It is implicitly invoked by the system.	A method is called by the programmer.
4.	The new keyword plays an important role in invoking the constructor.	Method calls are responsible for invoking methods.
5.	It has no return type. It can or cannot return any value to the caller.	So, it has a return type.
6.	The constructor name will always be the same as the class name.	We can use any name for the method name, such as addRow, addNum and subNumbers etc.
7.	A class can have more than one parameterized constructor. But constructors should have different parameters.	A class can also have more than one method with the same name but different in arguments and datatypes.

Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
1. #include <iostream>
2. using namespace std;
3. class Test
4. {
5.     private:
6.        int num;
7.     public:
8.        Test(): num(8){}
9.        void operator ++()        {
10.          num = num+2;
11.       }
12.       void Print() {
13.          cout<<"The Count is: "<<num;
14.       }
```

```
15. };
16. int main()
17. {
18.     Test tt;
19.     ++tt;  // calling of a function "void operator ++()"
20.     tt.Print();
21.     return 0;
22. }
```

Output:

```
The Count is: 10
```

Constructor overloading in Java

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task. Consider the following Java program, in which we have used different constructors in the class.

Example

```
1.  public class Student {
2.  //instance variables of the class
3.  int id;
4.  String name;
5.
6.  Student(){
7.  System.out.println("this a default constructor");
8.  }
9.
10. Student(int i, String n){
11. id = i;
12. name = n;
13. }
```

```
14.
15. public static void main(String[] args) {
16. //object creation
17. Student s = new Student();
18. System.out.println("\nDefault Constructor values: \n");
19. System.out.println("Student Id : "+s.id + "\nStudent Name : "+s.name);
20.
21. System.out.println("\nParameterized Constructor values: \n");
22. Student student = new Student(10, "David");
23. System.out.println("Student Id : "+student.id + "\nStudent Name : "+student.name);
24. }
25. }
```

Output:

```
this a default constructor
Default Constructor values:
Student Id : 0
Student Name : null
Parameterized Constructor values:
Student Id : 10
Student Name : David
```

Java Copy Constructor

Like C++, Java also supports the copy constructor. But in C++ it is created by default. While in Java we define copy constructor our own. In this section, we will learn the copy constructor in Java with an example. In Java, a constructor is the same as a method but the only difference is that the constructor has the same name as the class name. It is used to create an instance of the class. It is called automatically when we create an object of the class. It has no return type. Remember that a constructor cannot be abstract, final, synchronized, and static. We cannot override a constructor. It occupies some space in memory when it is called.

Types of Constructors

- Default Constructor
- Parameterized Constructor

Copy Constructor

In Java, a copy constructor is a special type of constructor that creates an object using another object of the same Java class. It returns a duplicate copy of an existing object of the class.

Use of Copy Constructor

We can use the copy constructor if we want to:

- Create a copy of an object that has multiple fields.
- Generate a deep copy of the heavy objects.
- Avoid the use of the Object.clone() method.

Advantages of Copy Constructor

- If a field declared as final, the copy constructor can change it.
- There is no need for typecasting.
- Its use is easier if an object has several fields.

Example of Copy Constructor

CopyConstructorExample.java

```
1. public class Fruit
2. {
3.  private double fprice;
4.  private String fname;
5.  //constructor to initialize roll number and name of the student
6.  Fruit(double fPrice, String fName)
7. {
8.  fprice = fPrice;
9.  fname = fName;
10. }
11. //creating a copy constructor
12. Fruit(Fruit fruit)
13. {
14. System.out.println("\nAfter invoking the Copy Constructor:\n");
15. fprice = fruit.fprice;
16. fname = fruit.fname;
```

```
17. }
18. //creating a method that returns the price of the fruit
19. double showPrice()
20. {
21. return fprice;
22. }
23. //creating a method that returns the name of the fruit
24. String showName()
25. {
26. return fname;
27. }
28. //class to create student object and print roll number and name of the student
29. public static void main(String args[])
30. {
31. Fruit f1 = new Fruit(399, "Ruby Roman Grapes");
32. System.out.println("Name of the first fruit: "+ f1.showName());
33. System.out.println("Price of the first fruit: "+ f1.showPrice());
34. //passing the parameters to the copy constructor
35. Fruit f2 = new Fruit(f1);
36. System.out.println("Name of the second fruit: "+ f2.showName());
37. System.out.println("Price of the second fruit: "+ f2.showPrice());
38. }
39. }
```

Output:

```
Name of the first fruit: Ruby Roman Grapes
Price of the first fruit: 399.0

After invoking the Copy Constructor:

Name of the second fruit: Ruby Roman Grapes
Price of the second fruit: 399.0
```

Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

It makes the code compact but complex to understand.

Syntax:

```
1. returntype methodname(){
2. //code to be executed
3. methodname();//calling same method
4. }
```

Java Recursion Example 1: Infinite times

```
1. public class RecursionExample1 {
2. static void p(){
3. System.out.println("hello");
4. p();
5. }
6.
7. public static void main(String[] args) {
8. p();
9. }
10. }
```

Output:

```
hello
hello
...
java.lang.StackOverflowError
```

Java Recursion Example 3: Factorial Number

```
1. public class RecursionExample3 {
2.     static int factorial(int n){
3.     if (n == 1)
4.           return 1;
5.         else
6.     return(n * factorial(n-1));7.
       }
8.
9. public static void main(String[] args) {
10. System.out.println("Factorial of 5 is: "+factorial(5));
11. }
12. }
```

Output:

```
Factorial of 5 is: 120
```

Garbage Collection in java

In **Java, garbage collection** is the process of managing memory, automatically. It finds the unused objects (that are no longer used by the program) and delete or remove them to free up the memory. The **garbage collection** mechanism uses several GC algorithms. The most popular algorithm that is used is **Mark and Sweep**.

Important Points About Garbage Collector

- It is controlled by a thread known as **Garbage Collector**.
- Java provides two methods **System.gc()** and **Runtime.gc()** that sends request to the JVM for garbage collection. Remember, it is not necessary that garbage collection will happen.
- Java programmer are free from memory management. We cannot force the garbage collector to collect the garbage, it depends on the JVM.
- If the Heap Memory is full, the JVM will not allow to create a new object and shows an error **java.lang.OutOfMemoryError**.
- When garbage collector removes object from the memory, first, the garbage collector thread calls the finalize() method of that object and then remove.

Important Points About Garbage Collector

- It is controlled by a thread known as **Garbage Collector**.
- Java provides two methods **System.gc()** and **Runtime.gc()** that sends request to the JVM for garbage collection. Remember, it is not necessary that garbage collection will happen.
- Java programmer are free from memory management. We cannot force the garbage collector to collect the garbage, it depends on the JVM.
- If the Heap Memory is full, the JVM will not allow to create a new object and shows an error **java.lang.OutOfMemoryError**.

Types of Garbage Collection

There are five types of garbage collection are as follows:

- **Serial GC:** It uses the mark and sweeps approach for young and old generations, which is minor and major GC.
- **Parallel GC:** It is similar to serial GC except that, it spawns N (the number of CPU cores in the system) threads for young generation garbage collection.
- **Parallel Old GC:** It is similar to parallel GC, except that it uses multiple threads for both generations.

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. }
```

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

Java String

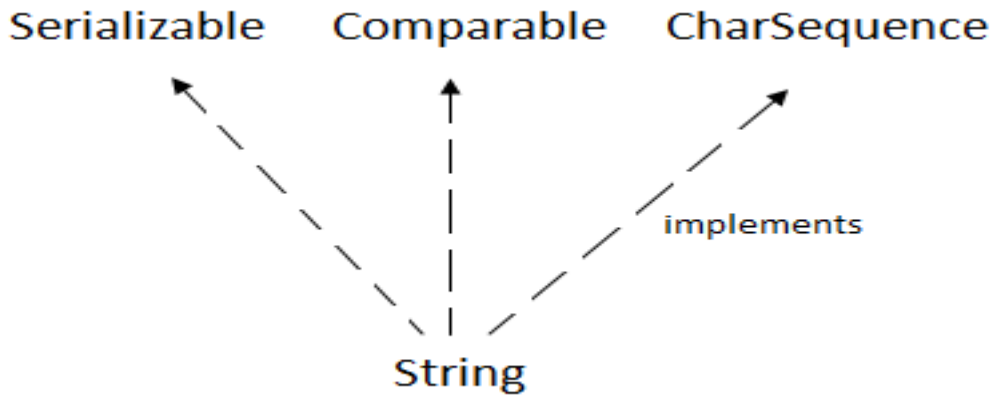
In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

```
1. char[] ch={'j','a','v','a'};
2. String s=new String(ch);
```

is same as:

```
1. String s="java";
```

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.



Java String Example

StringExample.java

```
1.  public class StringExample{
2.  public static void main(String args[]){
3.  String s1="java";//creating string by Java string literal
4.  char ch[]={'s','t','r','i','n','g','s'};
5.  String s2=new String(ch);//converting char array to string
6.  String s3=new String("example");//creating Java string by new keyword
7.  System.out.println(s1);
8.  System.out.println(s2);
9.  System.out.println(s3);
10. }}
```

Output:

```
java
strings
example
```

Immutable String in Java

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

```
1. class Testimmutablestring{
2.  public static void main(String args[]){
3.    String s="Sachin";
4.    s.concat(" Tendulkar");//concat() method appends the string at the end
5.  System.out.println(s);//will print Sachin because strings are immutable objects
6.  }
7. }
```

Output:

```
Sachin
```

Testimmutablestring1.java

```
1. class Testimmutablestring1{
2.  public static void main(String args[]){
3.    String s="Sachin";
4.    s=s.concat(" Tendulkar");
5.  System.out.println(s);
6.  }
7. }
```

Output:

```
Sachin Tendulkar
```

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:

CHAPTER-6

Inheritance in Java

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.

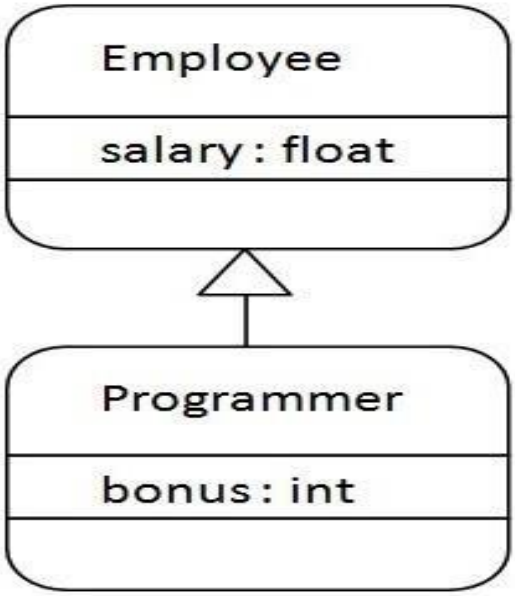
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

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.

Java Inheritance Example




```
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();
```

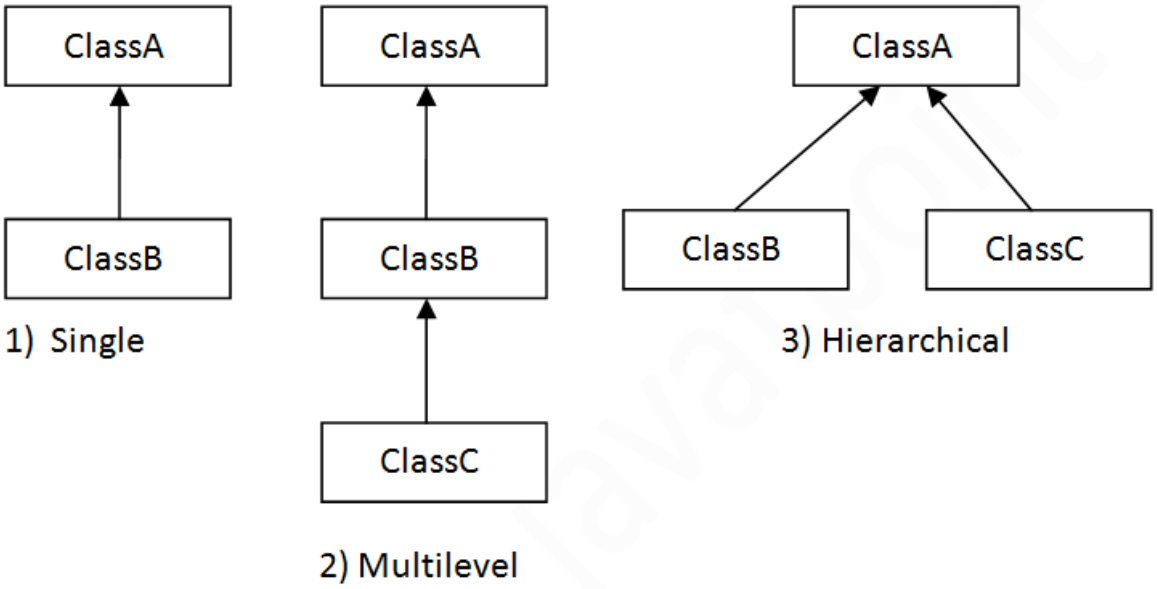
```
8.      System.out.println("Programmer salary is:"+p.salary);
9.      System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }
```

Output:

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

Types of inheritance in java

Single Inheritance



Example

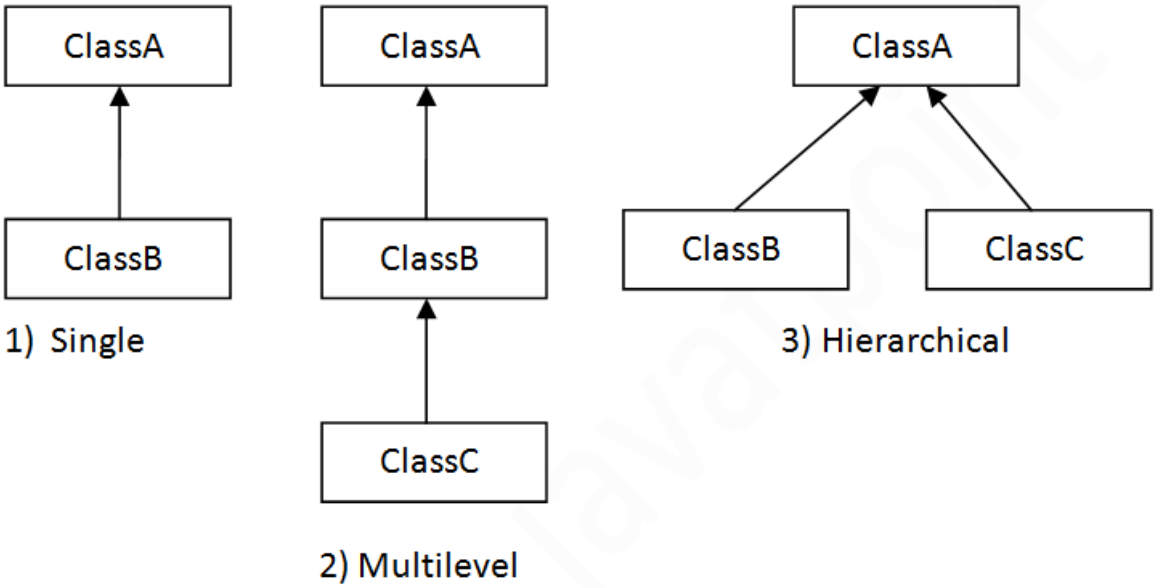
When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

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. }}
```

```
13. Output:
14. barking...
15. 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

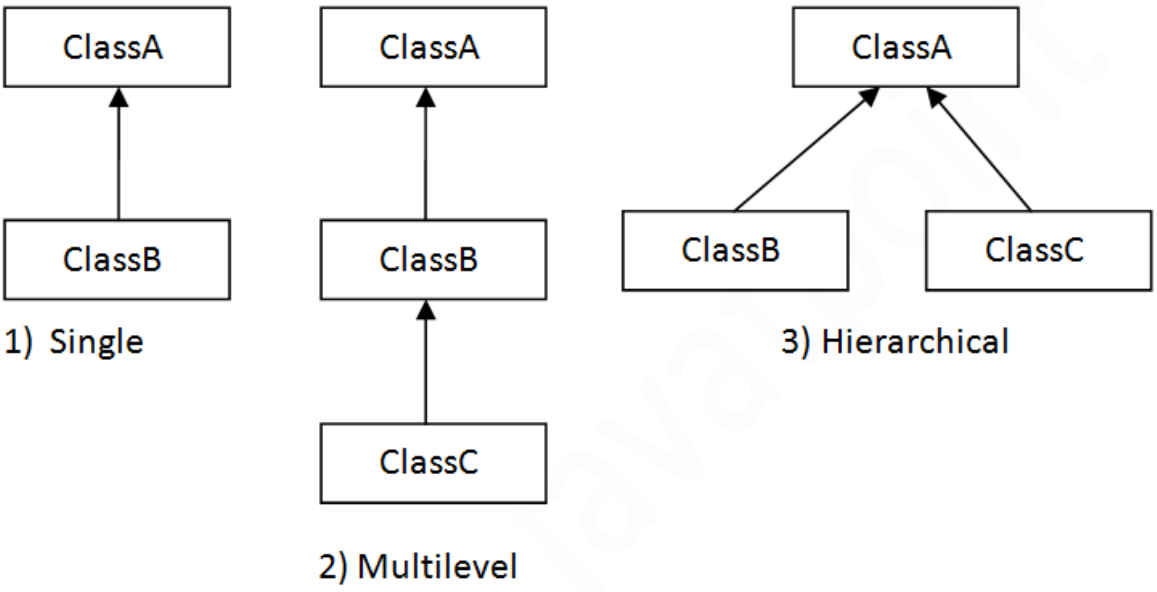
```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void bark(){System.out.println("barking...");}
6.  }
7.  class BabyDog extends Dog{
8.  void weep(){System.out.println("weeping...");}
9.  }
10. class TestInheritance2{
11. public static void 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. }}
17. Output:
18. meowing...
19. eating..
```

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

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...
```

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:

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.

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 behaviour of the method because its name differs.

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).

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. class Vehicle{
4.   //defining a method
5.   void run(){System.out.println("Vehicle is running");}
6. }
```

```
7.  //Creating a child class
8.  class Bike2 extends Vehicle{
9.    //defining the same method as in the parent class
10.   void run(){System.out.println("Bike is running safely");}
11.
12.   public static void main(String args[]){
13.   Bike2 obj = new Bike2();//creating object
14.   obj.run();//calling method
15.  }
16. }
```

Output:

```
Bike is running safely
```

```
1.  //Java Program to demonstrate the real scenario of Java Method Overriding
2.  //where three classes are overriding the method of a parent class.
3.  //Creating a parent class.
4.  class Bank{
5.  int getRateOfInterest(){return 0;}
6. }
7.  //Creating child classes.
8.  class SBI extends Bank{
9.  int getRateOfInterest(){return 8;}
10. }
11.
12. class ICICI extends Bank{
13. int getRateOfInterest(){return 7;}
14. }
15. class AXIS extends Bank{
16. int getRateOfInterest(){return 9;}
17. }
```

```
18. //Test class to create objects and call the methods
19. class Test2{
20. public static void main(String args[]){
21. SBI s=new SBI();
22. ICICI i=new ICICI();
23. AXIS a=new AXIS();
24. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
25. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
26. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
27. }
28. }
```

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

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).

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. }
```

Output:
running safely

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

How to declare an interface?

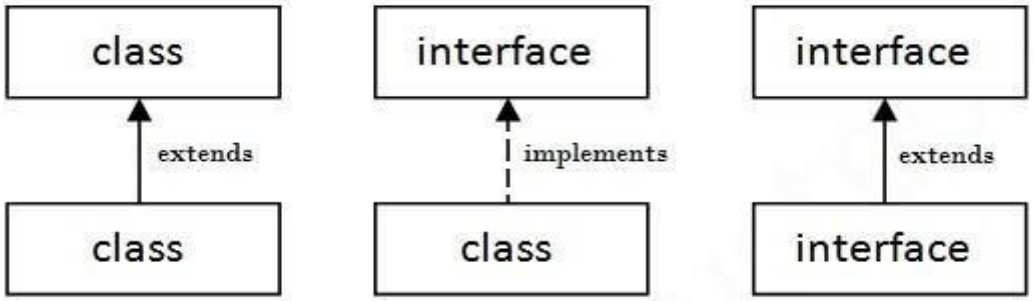
An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the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

Syntax:

```
1. interface <interface_name>{
2.
3.      // declare constant fields
4.      // declare methods that abstract
5.      // by default.
6. }
```

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.

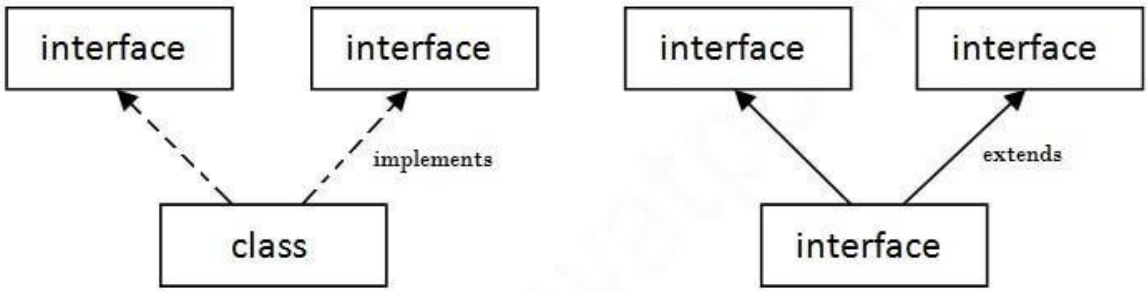
```
1. interface printable{
2. void print();
3. }
4. class A6 implements printable{
5. public void print(){System.out.println("Hello");}
6.
7. public static void main(String args[]){
8. A6 obj = new A6();
9. obj.print();
10. }
11. }
```

Output:

```
Hello
```

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

```
1.  interface Printable{
2.  void print();
3. }
4.  interface Showable{
5.  void show();
6.  }
7.  class A7 implements Printable,Showable{
8.  public void print(){System.out.println("Hello");}
9.  public void show(){System.out.println("Welcome");}
10.
11. public static void main(String args[]){
12. A7 obj = new A7();
13. obj.print();
14. obj.show();
15. }
16. }
```

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.

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 Inner Classes (Nested Classes)

Java inner class or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
1. class Java_Outer_class{
2.  //code
```

```
3.  class Java_Inner_class{
4.   //code
5.  }
6. }
```

Advantage of Java inner classes

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization**: It requires less code to write.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
    1. Member inner class
    2. Anonymous inner class
    3. Local inner class

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.

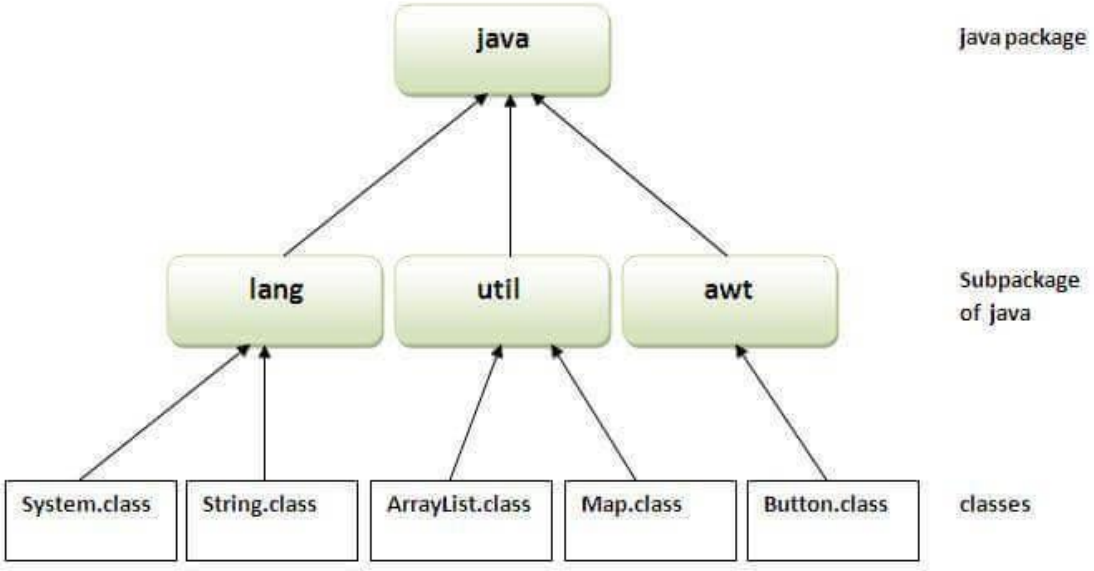
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

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. }
```

How to compile java package

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

```
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).

How to run java package program

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

To Compile: javac -d . Simple.java
To Run: java mypack.Simple
Output:Welcome to package

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.

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.

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

CHAPTER-7

Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

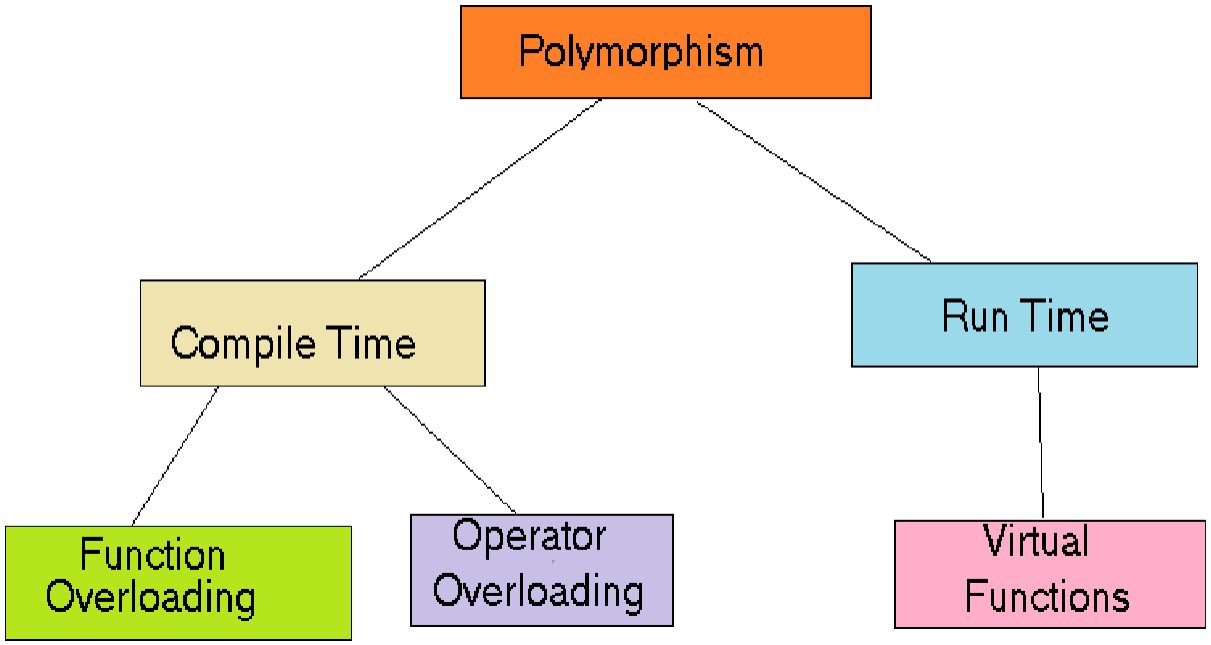
Real-life Illustration Polymorphism: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

What is Polymorphism in Java?

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, so it means many forms.

Types of Java polymorphism

In Java polymorphism is mainly divided into two types:



- Compile-time Polymorphism
- Runtime Polymorphism

Compile-Time Polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

Method Overloading

When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

Example 1:

```
// Java Program for Method overloading

// By using Different Types of Arguments


// Class 1

// Helper class

class Helper {


    // Method with 2 integer parameters

    static int Multiply(int a, int b)

    {


        // Returns product of integer numbers

        return a * b;

    }


    // Method 2

    // With same name but with 2 double parameters

    static double Multiply(double a, double b)

    {


        // Returns product of double numbers
```

```
        return a * b;
    }
}

// Class 2
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Calling method by passing
        // input as in arguments
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

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
```

Runtime Polymorphism in Java

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

```
1. class A{}
2. class B extends A{}
1. A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
1. interface I{}
2. class A{}
3. class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

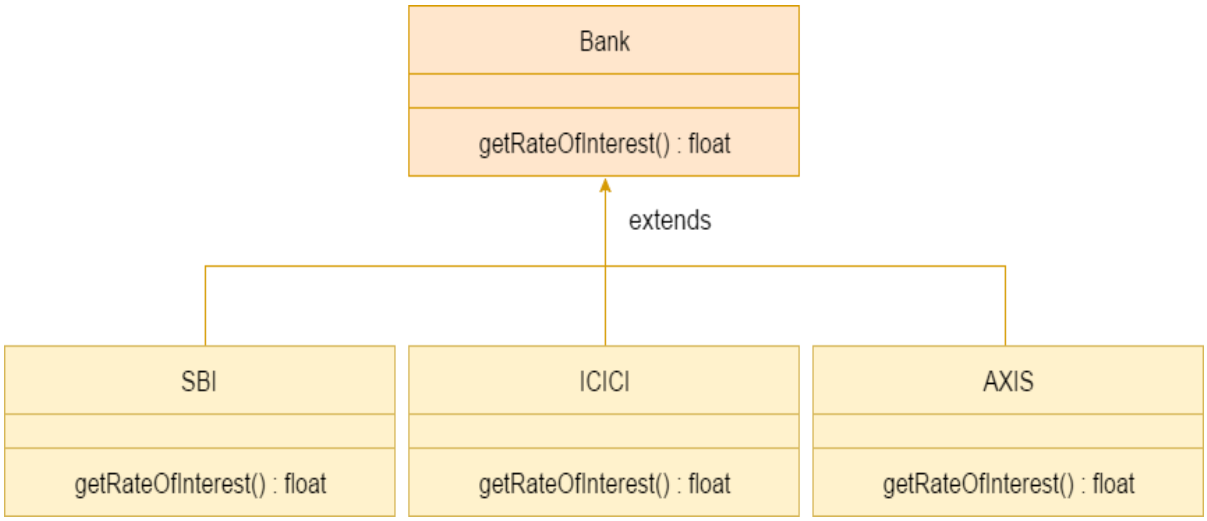
```
1. class Bike{
2.   void run(){System.out.println("running");}
3. }
4. class Splendor extends Bike{
5.   void run(){System.out.println("running safely with 60km");}
6.
7.   public static void main(String args[]){
8.     Bike b = new Splendor();//upcasting
9.     b.run();
10.  }
11. }
```

Output:

```
running safely with 60km.
```

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.




```
1.  class Bank{
2.  float getRateOfInterest(){return 0;}
3. }
4.  class SBI extends Bank{
5.  float getRateOfInterest(){return 8.4f;}
6. }
7.  class ICICI extends Bank{
8.  float getRateOfInterest(){return 7.3f;}
9. }
10. class AXIS extends Bank{
11. float getRateOfInterest(){return 9.7f;}
12. }
13. class TestPolymorphism{
14. public static void main(String args[]){
15. Bank b;
16. b=new SBI();
17. System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
18. b=new ICICI();
```

```
19. System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
20. b=new AXIS();
21. System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
22. }
23. }
```

Test it Now

Output:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Java Runtime Polymorphism Example: Shape

```
1.  class Shape{
2.  void draw(){System.out.println("drawing...");}
3.  }
4.  class Rectangle extends Shape{
5.  void draw(){System.out.println("drawing rectangle...");}
6.  }
7.  class Circle extends Shape{
8.  void draw(){System.out.println("drawing circle...");}
9.  }
10. class Triangle extends Shape{
11. void draw(){System.out.println("drawing triangle...");}
12. }
13. class TestPolymorphism2{
14. public static void main(String args[]){
15. Shape s;
16. s=new Rectangle();
17. s.draw();
18. s=new Circle();
19. s.draw();
20. s=new Triangle();
21. s.draw();
22. }
23. }
```

Output:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

Java Runtime Polymorphism Example: Animal

```
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. }
7. class Cat extends Animal{
8. void eat(){System.out.println("eating rat...");}
9. }
10. class Lion extends Animal{
11. void eat(){System.out.println("eating meat...");}
12. }
13. class TestPolymorphism3{
14. public static void main(String[] args){
15. Animal a;
16. a=new Dog();
17. a.eat();
18. a=new Cat();
19. a.eat();
20. a=new Lion();
21. a.eat();
22. }}
```

Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

```
1.  class Bike{
2.   int speedlimit=90;
3. }
4.  class Honda3 extends Bike{
5.   int speedlimit=150;
6.
7.   public static void main(String args[]){
8.    Bike obj=new Honda3();
9.    System.out.println(obj.speedlimit);//90
10. }
```

Output

90

Static Binding	Dynamic Binding
It takes place at compile time for which is referred to as early binding	It takes place at runtime so do it is referred to as late binding.
It uses overloading more precisely operator overloading method	It uses overriding methods.
It takes place using normal functions	It takes place using virtual functions
Static or const or private functions use real objects in static binding	Real objects use dynamic binding.

virtual class

Being an object-oriented programming language java supports features like inheritance, polymorphism, upcasting, etc. Therefore, OOPs in Java deals with objects, classes, and functions. A virtual function is one of a member function which facilitates run time polymorphism in Java. In this article, we will discuss the virtual function in Java.

Definition: A virtual function is not any special function, but it is a member function that facilitates the method overriding mechanism. That means, in OOPs, a virtual function of the parent class is a function that can be overridden by a child class which has the same type but with different functionality.

Syntax: For Virtual Function in Java, you should follow the basic syntax of java with annotations. To implement the overriding mechanism for the virtual function, @Override annotation may be used here to specifically point out which virtual function we want to override. Although it is not mandatory.

How Virtual Function Works in Java?

Now let us see how virtual function works. When we call an overridden method of child class through its parent type reference, then the type or reference of the object indicates which method will be invoked. The conclusion of this decision occurs during runtime after the compilation. So, the functionality of the virtual function is overridden by the inherited child class of the same type.Some Points Regarding Virtual Function:

- Functions of child and parent class must have the same name and have the same parameter.
- IS-A relationship is mandatory (inheritance).
- A virtual function cannot be private as we cannot override private methods of the parent class.
- A virtual function cannot be stated as Final, as we cannot override Final methods.
- A virtual function cannot be stated as Static, as we cannot override static methods.

CHAPTER-8

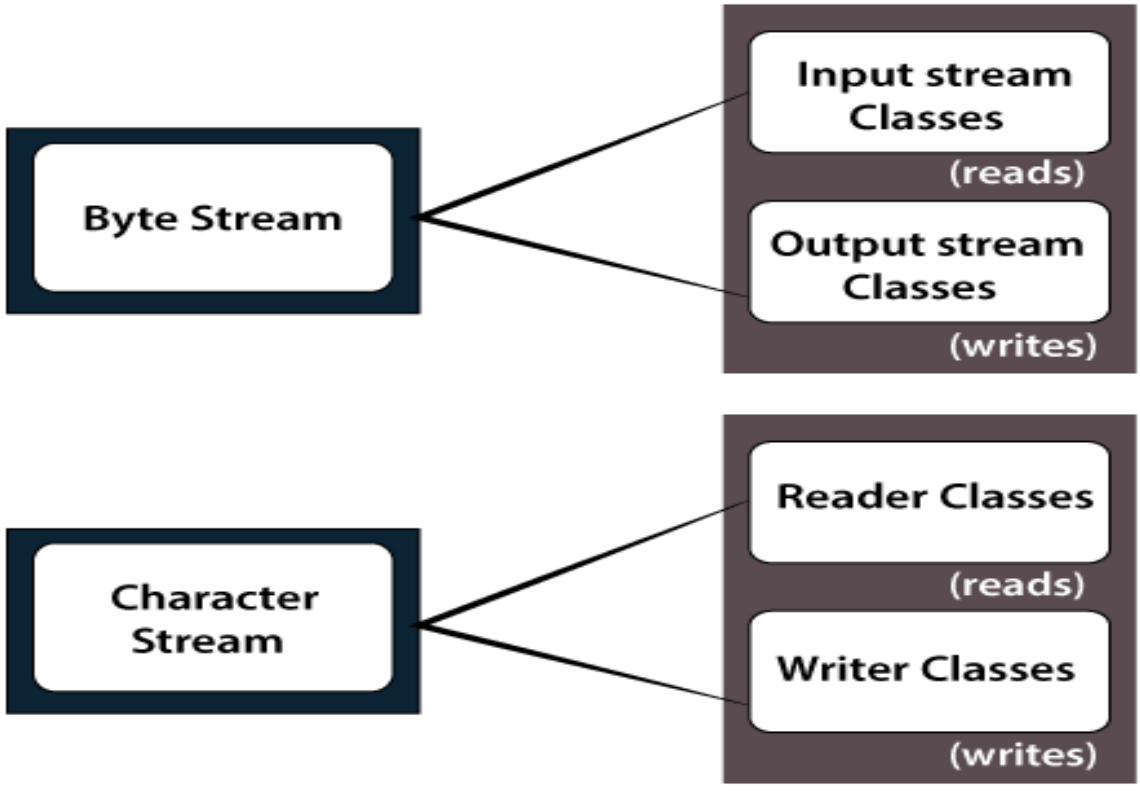
File Handling in Java

In Java, a **File** is an abstract data type. A named location used to store related information is known as a **File**. There are several **File Operations** like **creating a new File, getting information about File, writing into a File, reading from a File** and **deleting a File**.

Before understanding the File operations, it is required that we should have knowledge of **Stream** and **File methods**. If you have knowledge about both of them, you can skip it.

Stream

A series of data is referred to as **a stream**. In Java, **Stream** is classified into two types, i.e., **Byte Stream** and **Character Stream**.



Brief classification of I/O streams

Byte Stream

Byte Stream is mainly involved with byte data. A file handling process with a byte stream is a process in which an input is provided and executed with the byte data.

Character Stream

Character Stream is mainly involved with character data. A file handling process with a character stream is a process in which an input is provided and executed with the character data.

Java File Class Methods

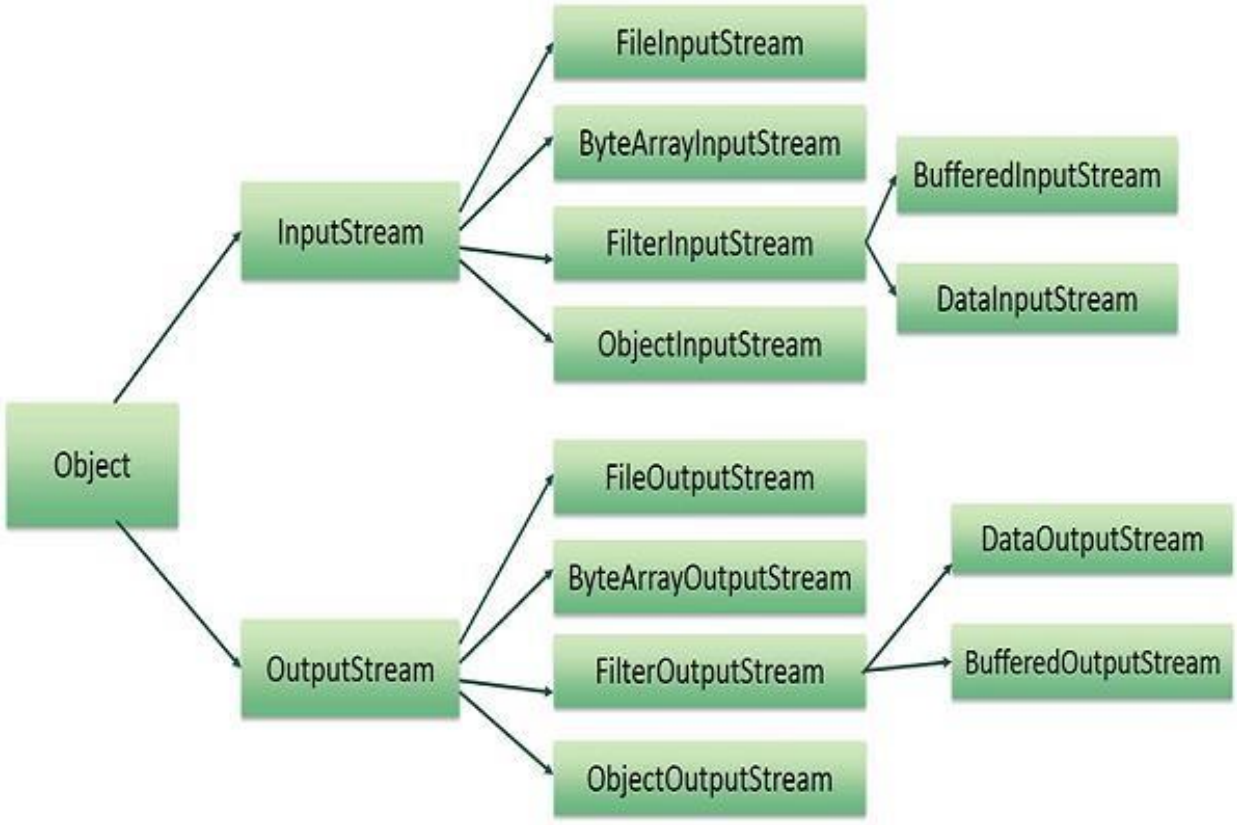
S.No.	Method	Return Type	Description
1.	canRead()	Boolean	The **canRead()** method is used to check whether we ca read the data of the file or not.
2.	createNewFile()	Boolean	The **createNewFile()** method is used to create a new empty file.
3.	canWrite()	Boolean	The **canWrite()** method is used to check whether we ca write the data into the file or not.
4.	exists()	Boolean	The **exists()** method is used to check whether the specified file is present or not.
5.	delete()	Boolean	The **delete()** method is used to delete a file.
6.	getName()	String	The **getName()** method is used to find the file name.
7.	getAbsolutePath()	String	The **getAbsolutePath()** method is used to get th absolute pathname of the file.
8.	length()	Long	The **length()** method is used to get the size of the file i bytes.
9.	list()	String[]	The **list()** method is used to get an array of the files available in the directory.
10.	mkdir()	Boolean	The **mkdir()** method is used for creating a new directory

Stream In Java

Introduced in Java 8, Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result. The features of Java stream are –

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.

- Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.



1. Input Stream:
The Java InputStream class is the superclass of all input streams. The input stream is used to read data from numerous input devices like the keyboard, network, etc. InputStream is an abstract class, and because of this, it is not useful by itself. However, its subclasses are used to read data.

There are several subclasses of the InputStream class, which are as follows:

1. AudioInputStream
2. ByteArrayInputStream
3. FileInputStream
4. FilterInputStream
5. StringBufferInputStream
6. ObjectInputStream

Creating an InputStream

// Creating an InputStream

InputStream obj = new FileInputStream ();

Methods of InputStream

S No.	Method	Description
1	read()	Reads one byte of data from the input stream.
2	read(byte[] array)()	Reads byte from the stream and stores that byte in the specified array.
3	mark()	It marks the position in the input stream until the data has been read.
4	available()	Returns the number of bytes available in the input stream.
5	markSupported()	It checks if the mark() method and the reset() method is supported in the stream.
6	reset()	Returns the control to the point where the mark was set inside the stream.
7	skips()	Skips and removes a particular number of bytes from the input stream.
8	close()	Closes the input stream.

2. Output Stream:

The output stream is used to write data to numerous output devices like the monitor, file, etc. OutputStream is an abstract superclass that represents an output stream. OutputStream is an abstract class and because of this, it is not useful by itself. However, its subclasses are used to write data.

There are several subclasses of the OutputStream class which are as follows:

1. ByteArrayOutputStream
2. FileOutputStream
3. StringBufferOutputStream
4. ObjectOutputStream
5. DataOutputStream
6. PrintStream

Creating an OutputStream

```
// Creating an OutputStream

OutputStream obj = new FileOutputStream();
```

Methods of OutputStream

S. No.	Method	Description
1.	write()	Writes the specified byte to the output stream.
2.	write(byte[] array)	Writes the bytes which are inside a specific array to the output stream.
3.	close()	Closes the output stream.
4.	flush()	Forces to write all the data present in an output stream to the destination.

Based on the data type, there are two types of streams :

1. Byte Stream:

This stream is used to read or write byte data. The byte stream is again subdivided into two types which are as follows:

- **Byte Input Stream:** Used to read byte data from different devices.
- **Byte Output Stream:** Used to write byte data to different devices.

2. Character Stream:

This stream is used to read or write character data. Character stream is again subdivided into 2 types which are as follows:

- **Character Input Stream:** Used to read character data from different devices.
- **Character Output Stream:** Used to write character data to different devices.

Java File Class Methods

The following table depicts several File Class methods:

Method Name	Description	Return Type
canRead()	It tests whether the file is readable or not.	Boolean
canWrite()	It tests whether the file is writable or not.	Boolean
createNewFile()	It creates an empty file.	Boolean
delete()	It deletes a file.	Boolean
exists()	It tests whether the file exists or not.	Boolean
length()	Returns the size of the file in bytes.	Long
getName()	Returns the name of the file.	String
list()	Returns an array of the files in the directory.	String[]
mkdir()	Creates a new directory	Boolean
getAbsolutePath()	Returns the absolute pathname of the file.	String

File Operations

We can perform the following operation on a file:

- Create a File
- Get File Information
- Write to a File
- Read from a File
- Delete a File

Create a File

Create a File operation is performed to create a new file. We use the **createNewFile()** method of file. The **createNewFile()** method returns true when it successfully creates a new file and returns false when the file already exists.

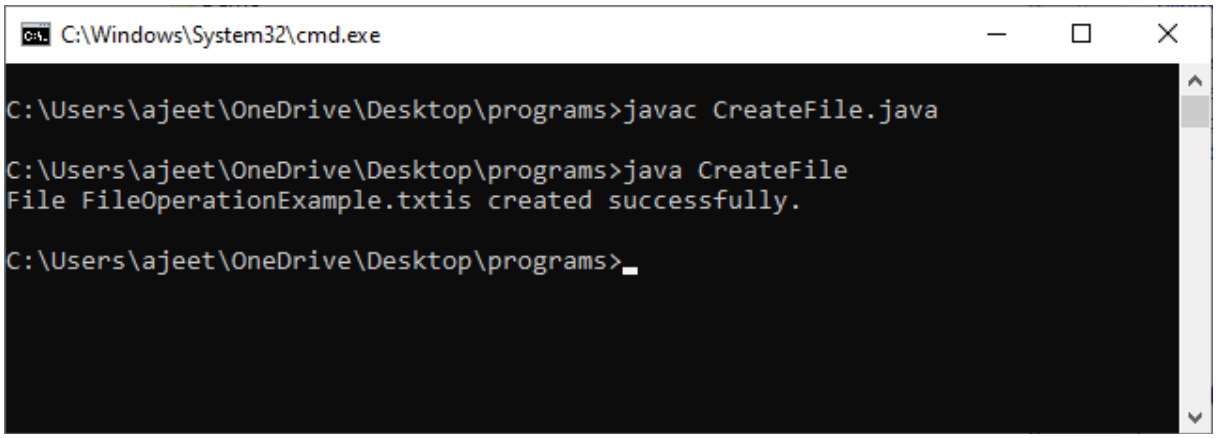
Let's take an example of creating a file to understand how we can use the **createNewFile()** method to perform this operation.

CreateFile.java

```
1. // Importing File class
2. import java.io.File;
3. // Importing the IOException class for handling errors
4. import java.io.IOException;
5.  class CreateFile {
6.             public static void main(String args[]) {
7.             try {
8.                   // Creating an object of a file
9.                   File f0 = new File("D:FileOperationExample.txt");
10.                  if (f0.createNewFile()) {
11.                        System.out.println("File " + f0.getName() + " is created successfully.");

12.                  } else {
13.                        System.out.println("File is already exist in the directory.");
14.                  }
15.                } catch (IOException exception) {
16.                      System.out.println("An unexpected error is occurred.");
17.                      exception.printStackTrace();
18.             }
19.       }
20. }
```

Output:



Get File Information

The operation is performed to get the file information. We use several methods to get the information about the file like name, absolute path, is readable, is writable and length.

Let's take an example to understand how to use file methods to get the information of the file.

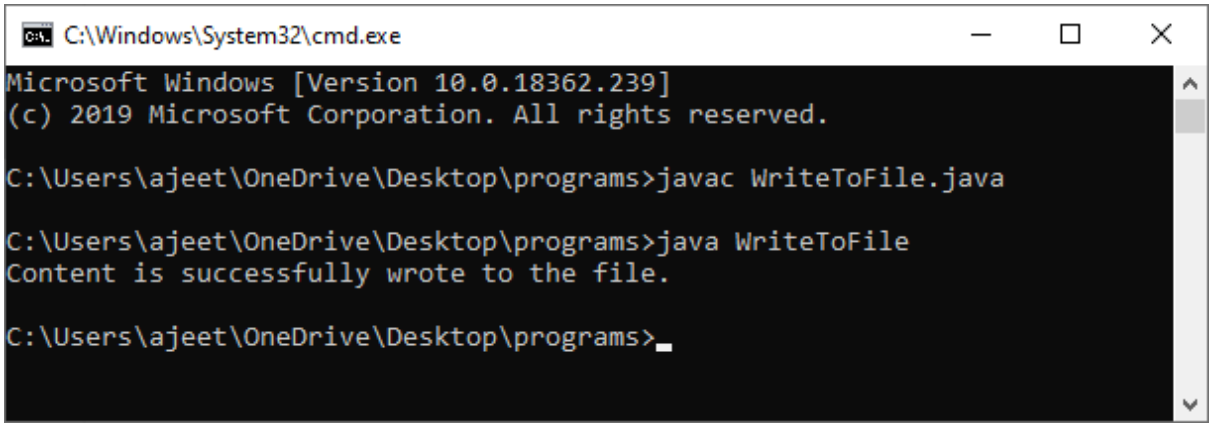
FileInfo.java

WriteToFile.java

```
1.  // Importing the FileWriter class
2.  import java.io.FileWriter;
3.
4.  // Importing the IOException class for handling errors
5.  import java.io.IOException;
6.
7.  class WriteToFile {
8.     public static void main(String[] args) {
9.
10.    try {
11.       FileWriter fwrite = new FileWriter("D:FileOperationExample.txt");
12.       // writing the content into the FileOperationExample.txt file
13.       fwrite.write("A named location used to store related information is referred to as a File."
    );
14.14.
15.       // Closing the stream
16.       fwrite.close();
17.       System.out.println("Content is successfully wrote to the file.");
```

```
18.     } catch (IOException e) {
19.         System.out.println("Unexpected error occurred");
20.         e.printStackTrace();
21.         }
22.     }
23. }
```

Output:



Read from a File

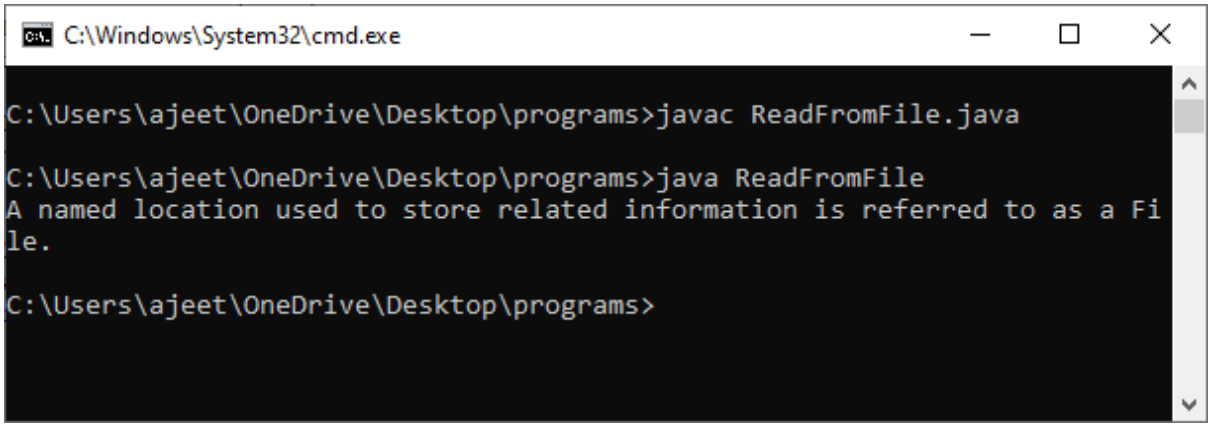
The next operation which we can perform on a file is **"read from a file"**. In order to write data into a file, we will use the **Scanner** class. Here, we need to close the stream using the **close()** method. We will create an instance of the Scanner class and use the **hasNextLine ()** method **nextLine()** method to get data from the file. Let's take an example to understand how we can read data from a file.

ReadFromFile.java

```
1.  // Importing the File class
2.  import java.io.File;
3.  // Importing FileNotFoundException class for handling errors
4.  import java.io.FileNotFoundException;
5.  // Importing the Scanner class for reading text files
6.  import java.util.Scanner;
7.
8.  class ReadFromFile {
9.      public static void main(String[] args) {
10.         try {
```

```
11.         // Create f1 object of the file to read data
12.         File f1 = new File("D:FileOperationExample.txt");
13.         Scanner dataReader = new Scanner(f1);
14.         while (dataReader.hasNextLine()) {
15.             String fileData = dataReader.nextLine();
16.             System.out.println(fileData);
17.         }
18.         dataReader.close();
19.       } catch (FileNotFoundException exception) {
20.         System.out.println("Unexcpected error occurred!");
21.         exception.printStackTrace();
22.       }
23.   }
24. }
```

Output:



Delete a File

The next operation which we can perform on a file is **"deleting a file"**. In order to delete a file, we will use the **delete()** method of the file. We don't need to close the stream using the **close()** method because for deleting a file, we neither use the FileWriter class nor the Scanner class.

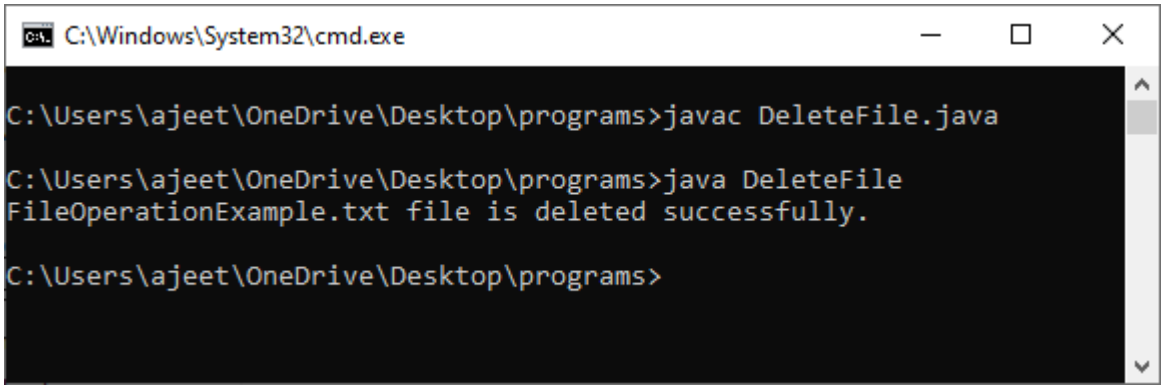
Let's take an example to understand how we can write data into a file.

DeleteFile.java

```
1. // Importing the File class
2. import java.io.File;
```

```
3.  class DeleteFile {
4.    public static void main(String[] args) {
5.      File f0 = new File("D:FileOperationExample.txt");
6.      if (f0.delete()) {
7.        System.out.println(f0.getName()+ " file is deleted successfully.");
8.      } else {
9.        System.out.println("Unexpected error found in deletion of the file.");
10.     }
11.   }
12. }
```

Output:



How to handle error

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

CHAPTER-9

Class templates & Function Templates

Template:

TEMPLATE Method is a behavioral design pattern that allows you to defines a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm's structure.

1. One of the major features of Java Generics is that It handles type checking during instantiation andgenerates byte-code equivalent to non-generic code.
   The compiler of Java checks type before instantiation, that in turn makes the implementation ofGeneric type-safe. Meanwhile, in C++, templates know nothing about types.
2. If Generics is applied in a class, then it gets Applied to classes and methods within classes.
3. Another major factor that leads to the use of generics in Java is because it allows you to eliminatedowncasts.
4. Instantiating a generic class has no runtime overhead over using an equivalent class that uses asspecific *object* rather than a generic type of *T*.

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.

While building large-scale projects, we need the code to be compatible with any kind of data which is provided to it. That is the place where your written code stands above that of others. Here what we meant is to make the code you write be generic to any kind of data provided to the program regardless of its data type. This is where Generics in Java and the similar in C++ named Template come in handy. While both have similar functionalities, but they differ in a few places

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.

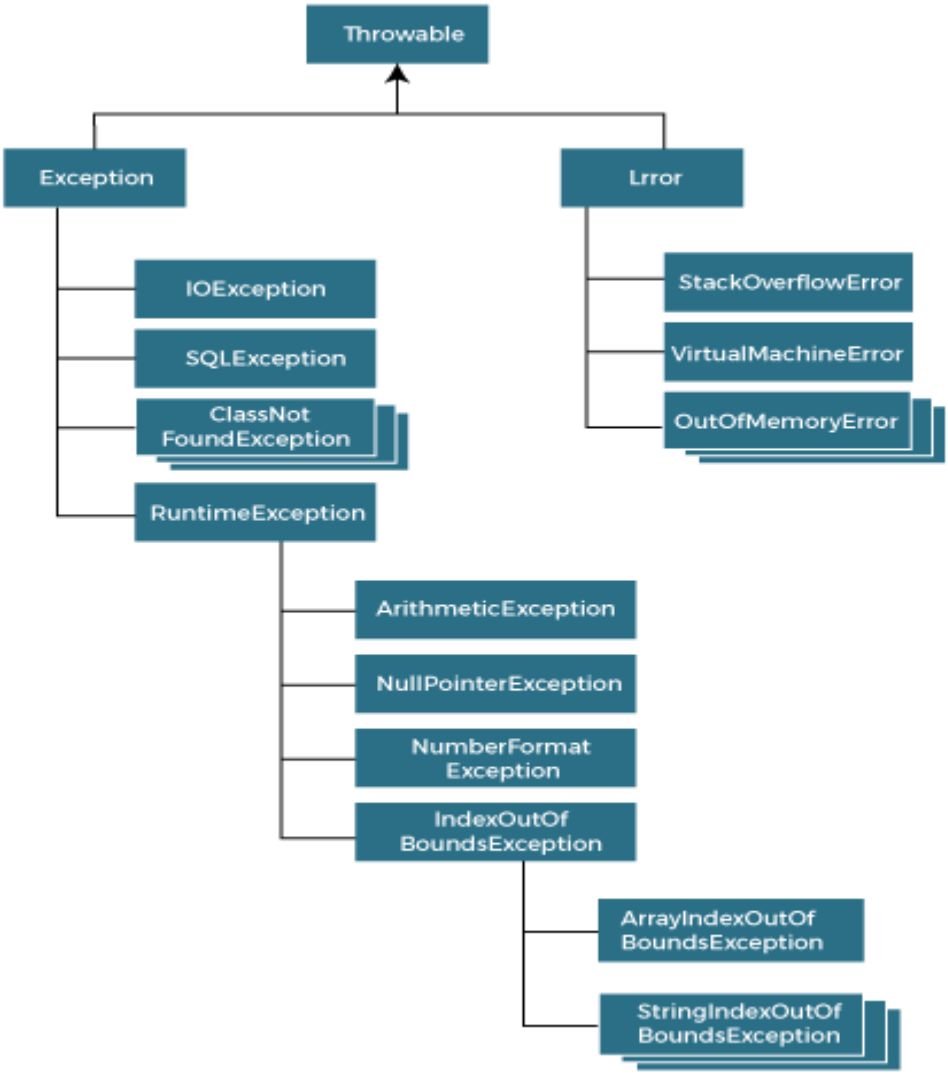
Advantage of Exception Handling

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;

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

Differences between Checked and Unchecked Exceptions in Java

S.No.	Checked Exception	Unchecked Exception
1.	Checked exceptions happen at compile time when the source code is transformed into an executable code.	Unchecked exceptions happen at runtime when the executable program starts running.
2.	The checked exception is checked by the compiler.	These types of exceptions are not checked by the compiler.
3.	Checked exceptions can be created manually.	They can also be created manually.
4.	This exception is counted as a sub-class of the class.	This exception happens in runtime, an hence it is not included in the exceptio class.
5.	Java Virtual Machine requires the exception to to be caught or handled.	Java Virtual Machine does not need th exception to be caught or handled.

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 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...
```

Java 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.

Internal Working of Java try-catch block

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Problem without exception handling

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

Example 1

TryCatchExample1.java

```
1. public class TryCatchExample1 {
2.
3.     public static void main(String[] args) {
4.
5.         int data=50/0; //may throw exception
6.
7.         System.out.println("rest of the code");
8.
9.     }
10.
11. }
```

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

Solution by exception handling

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

Example 2

TryCatchExample2.java

```
1. public class TryCatchExample2 {
2.
3.     public static void main(String[] args) {
4.         try
5.         {
6.         int data=50/0; //may throw exception
7.         }
8.             //handling the exception
9.         catch(ArithmeticException e)
10.        {
11.            System.out.println(e);
12.        }
13.        System.out.println("rest of the code");
```

```
14.     }
15.
16. }
```

Output:

java.lang.ArithmeticException: / by zero

rest of the code

Java Catch Multiple Exceptions

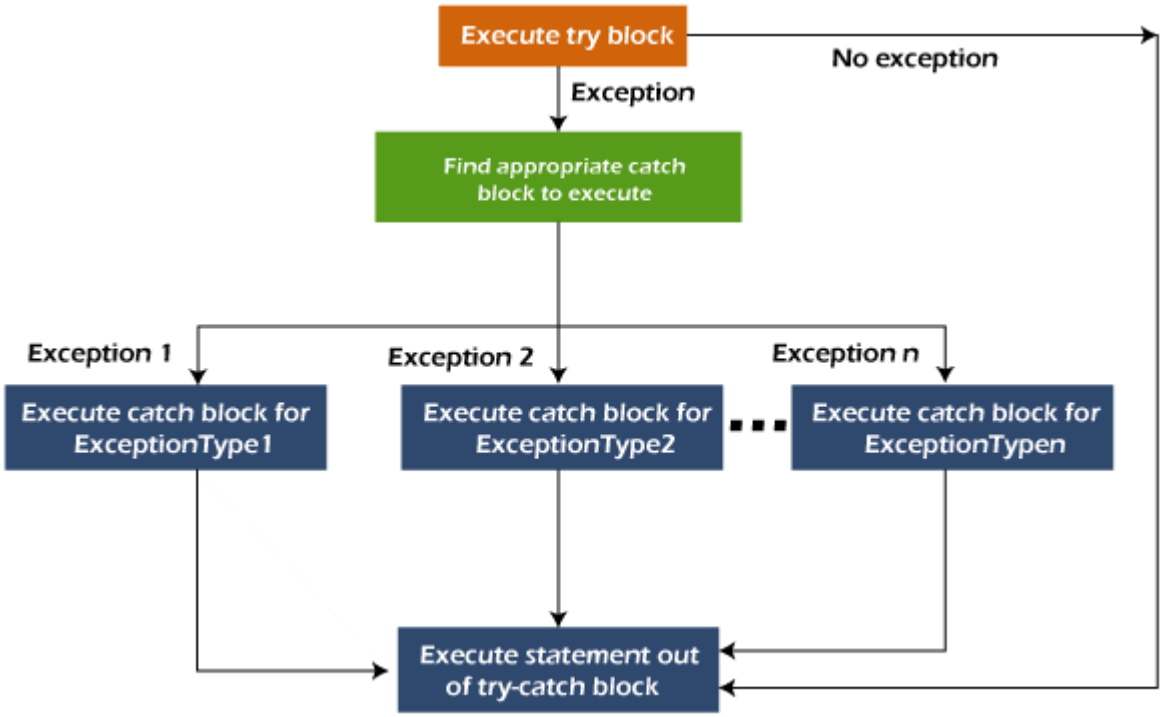
Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Flowchart of Multi-catch Block



Example 1

Let's see a simple example of java multi-catch block.

MultipleCatchBlock1.java

```
1. public class MultipleCatchBlock1 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.             a[5]=30/0;
8.            }
9.            catch(ArithmeticException e)
10.              {
11.               System.out.println("Arithmetic Exception occurs");
12.              }
13.           catch(ArrayIndexOutOfBoundsException e)
14.              {
```

```
15.                 System.out.println("ArrayIndexOutOfBounds Exception occurs");
16.               }
17.            catch(Exception e)
18.               {
19.                System.out.println("Parent Exception occurs");
20.               }
21.            System.out.println("rest of the code");
22.    }
23. }
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 2

MultipleCatchBlock2.java

```
1. public class MultipleCatchBlock2 {
2.
3.     public static void main(String[] args) {
4.
5.           try{
6.               int a[]=new int[5];
7.
8.               System.out.println(a[10]);
9.               }
10.              catch(ArithmeticException e)
11.                 {
12.                  System.out.println("Arithmetic Exception occurs");
13.                 }
14.              catch(ArrayIndexOutOfBoundsException e)
15.                 {
16.                  System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.                 }
```

```
18.             catch(Exception e)
19.               {
20.                System.out.println("Parent Exception occurs");
21.               }
22.             System.out.println("rest of the code");
23.     }
24. }
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

MultipleCatchBlock3.java

```
1. public class MultipleCatchBlock3 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.             a[5]=30/0;
8.             System.out.println(a[10]);
9.            }
10.           catch(ArithmeticException e)
11.             {
12.              System.out.println("Arithmetic Exception occurs");
13.             }
14.           catch(ArrayIndexOutOfBoundsException e)
15.             {
16.              System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.             }
18.           catch(Exception e)
19.             {
20.              System.out.println("Parent Exception occurs");
```

```
21.                 }
22.             System.out.println("rest of the code");
23.     }

    }
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Usage of Java finally

Let's see the different cases where Java finally block can be used.

Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

TestFinallyBlock.java

```
1. class TestFinallyBlock {
2.   public static void main(String args[]){
3.   try{
4. //below code do not throw any exception
5.    int data=25/5;
6.   System.out.println(data);
7.   }
8. //catch won't be executed
9.   catch(NullPointerException e){
10. System.out.println(e);
11. }
12. //executed regardless of exception occurred or not
13.  finally {
14. System.out.println("finally block is always executed");
15. }
16.
17. System.out.println("rest of phe code...");
18.  }
19. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

Case 2: When an exception occurr but not handled by the catch block

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

TestFinallyBlock1.java

```
1.  public class TestFinallyBlock1{
2.      public static void main(String args[]){
3.
4.      try {
5.
6.        System.out.println("Inside the try block");
7.
8.        //below code throws divide by zero exception
9.       int data=25/0;
10.      System.out.println(data);
11.     }
12.     //cannot handle Arithmetic type exception
13.     //can only accept Null Pointer type exception
14.     catch(NullPointerException e){
15.       System.out.println(e);
16.     }
17.
18.     //executes regardless of exception occured or not
19.     finally {
20.       System.out.println("finally block is always executed");
21.     }
22.
23.     System.out.println("rest of the code...");
24.     }
25.   }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

Case 3: When an exception occurs and is handled by the catch block

Example:

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

TestFinallyBlock2.java

```
1.  public class TestFinallyBlock2{
2.      public static void main(String args[]){
3.
4.      try {
5.
6.        System.out.println("Inside try block");
7.
8.        //below code throws divide by zero exception
9.       int data=25/0;
10.      System.out.println(data);
11.     }
12.
13.     //handles the Arithmetic Exception / Divide by zero exception
14.     catch(ArithmeticException e){
15.       System.out.println("Exception handled");
16.       System.out.println(e);
17.     }
18.
19.     //executes regardless of exception occured or not
20.     finally {
21.       System.out.println("finally block is always executed");
22.     }
23.
24.     System.out.println("rest of the code...");
25.     }
26.   }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Difference between throw and throws in Java

The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.

Java throw Example

TestThrow.java

```
1. public class TestThrow {
2.     //defining a method
3.     public static void checkNum(int num) {
4.         if (num < 1) {
5.         throw new ArithmeticException("\nNumber is negative, cannot calculate square");6.
           }
7.         else {
8.         System.out.println("Square of " + num + " is " + (num*num));9.
           }
10.    }
11.    //main method
12.    public static void main(String[] args) {
13.            TestThrow obj = new TestThrow();
14.            obj.checkNum(-3);
15.            System.out.println("Rest of the code..");
16.    }
17. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
        at TestThrow.checkNum(TestThrow.java:6)
        at TestThrow.main(TestThrow.java:16)
```

Java throws Example

TestThrows.java

```
1. public class TestThrows {
2.     //defining a method
3.     public static int divideNum(int m, int n) throws ArithmeticException {
4.         int div = m / n;
```

```
5.          return div;
6.      }
7.      //main method
8.      public static void main(String[] args) {
9.          TestThrows obj = new TestThrows();
10.         try {
11.             System.out.println(obj.divideNum(45, 0));
12.         }
13.         catch (ArithmeticException e){
14.             System.out.println("\nNumber cannot be divided by 0");
15.         }
16.
17.         System.out.println("Rest of the code..");
18.     }
19. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrows

Number cannot be divided by 0
Rest of the code..
```

Java throw and throws Example

TestThrowAndThrows.java

```
1. public class TestThrowAndThrows
2. {
3.     // defining a user-defined method
4.     // which throws ArithmeticException
5.     static void method() throws ArithmeticException
6.     {
7.         System.out.println("Inside the method()");
```

```
8.      throw new ArithmeticException("throwing ArithmeticException");9.
        }
10.     //main method
11.     public static void main(String args[])
12.     {
13.        try
14.        {
15.           method();
16.        }
17.        catch(ArithmeticException e)
18.        {
19.           System.out.println("caught in main() method");
20.        }
21.     }
22. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrowAndThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrowAndThrows
Inside the method()
caught in main() method
```

Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

Sr. no.	Key	final	finally	finalize

1.	**Definition**	**final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.**	**finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.**	**finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.**
2.	**Applicable to**	**Final keyword is used with the classes, methods and variables.**	**Finally block is always related to the try and catch block in exception handling.**	**finalize() method is used with the objects.**
3.	**Functionality**	**(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.**	**(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block**	**finalize method performs the cleaning activities with respect to the object before its destruction.**
4.	**Execution**	**Final method is executed only when we call it.**	**Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.**	**finalize method is executed just before the object is destroyed.**

Java final Example

Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

FinalExampleTest.java

```
1. public class FinalExampleTest {
2.     //declaring final variable
3.     final int age = 18;
```

```
4.      void display() {
5.
6.      // reassigning value to age variable
7.      // gives compile time error
8.      age = 55;
9.      }
10.
11.     public static void main(String[] args) {
12.
13.     FinalExampleTest obj = new FinalExampleTest();
14.     // gives compile time error
15.     obj.display();
16.     }
17. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac FinalExampleTest.java
FinalExampleTest.java:10: error: cannot assign a value to final variable age
        age = 55;
        ^
1 error
```

Java finally Example

Let's see the below example where the Java code throws an exception and the catch block handles that exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

FinallyExample.java

```
1.  public class FinallyExample {
2.      public static void main(String args[]){
3.      try {
4.        System.out.println("Inside try block");
5.      // below code throws divide by zero exception
6.       int data=25/0;
7.      System.out.println(data);
8.      }
9.      // handles the Arithmetic Exception / Divide by zero exception
10.     catch (ArithmeticException e){
```

```
11.         System.out.println("Exception handled");
12.         System.out.println(e);
13.       }
14.       // executes regardless of exception occurred or not
15.       finally {
16.         System.out.println("finally block is always executed");
17.       }
18.       System.out.println("rest of the code...");
19.       }
20.    }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>java FinallyExample.java
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Java finalize Example

FinalizeExample.java

```
1.  public class FinalizeExample {
2.     public static void main(String[] args)
3.     {
4.        FinalizeExample obj = new FinalizeExample();
5.        // printing the hashcode
6.        System.out.println("Hashcode is: " + obj.hashCode());
7.        obj = null;
8.        // calling the garbage collector using gc()
9.        System.gc();
10.       System.out.println("End of the garbage collection");
11.    }
12.   // defining the finalize method
13.    protected void finalize()
14.    {
15.       System.out.println("Called the finalize() method");
```

16. }
17. }

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac FinalizeExample.java
Note: FinalizeExample.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\Anurati\Desktop\abcDemo>java FinalizeExample
Hashcode is: 746292446
End of the garbage collection
Called the finalize() method
```

Exception Handling with Method Overriding in Java

There are many rules if we talk about method overriding with exception handling.

Some of the rules are listed below:

- **If the superclass method does not declare an exception**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.