Index

Java

1. Basics of Java

Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java.

Java is a **programming language** and a **platform**.
Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

Java is a high level, robust, object-oriented, class-based, concurrent and secure programming language.

Types of Java Applications

1. **Standalone Application** (desktop or window-based applications)
2. **Web Application** (runs on the server side)

3. **Enterprise Application** (distributed in nature, such as banking applications)
4. **Mobile Application** (created for mobile devices)

Java Editions/Platforms

1. **Java SE (Java Standard Edition)** → It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, string, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.
2. **Java EE (Java Enterprise Edition)** → used to develop web and enterprise applications. It is built on top of the Java SE platform.
3. **Java ME (Java Micro Edition)** → It is dedicated to mobile applications
4. **JavaFX** → It is used to develop rich internet applications

Features of Java

1. **Simple** (Java is very easy to learn, and its syntax is simple, clean and easy to understand.)
2. **Object-oriented** (Object, class based)
3. **Portable** (Java is portable because it facilitates you to carry the Java bytecode to any platform.)
4. **Platform Independent** (The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:- Runtime Environment and API.

   Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because

it can be run on multiple platforms, i.e., Write Once and Run Anywhere.)

5. **Secured** (Java is secured because it has no explicit pointer, Java Programs run inside a virtual machine sandbox, Classloader to load classes, Bytecode Verifier to verify code, Security Manager.)
6. **Robust** (Java is robust because:
   - It uses strong memory management.
   - There is a lack of pointers that avoids security problems.
   - Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
   - There are exception handling and the type checking mechanism in Java. All these points make Java robust.)
7. **Architecture Neutral** (Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.)
8. **Interpreted**
9. **High Performance** (Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.)
10. **Multithreaded** (Java programs deals with many tasks at once by defining multiple threads.)
11. **Distributed**

12. **Dynamic** (Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++. Java supports dynamic compilation and automatic memory management (garbage collection).)

C++ vs Java

Platform-independent	C++ is platform-dependent.	Java is platform-independent.

Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces.
Operator Overloading	C++ supports operator overloading	Java doesn't support operator overloading.
Pointers	C++ supports pointers You can write a pointer program in C++.	Java supports pointers internally. However, you can't write the pointer program in java. It means java has

		restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.

Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.

Differences between JDK, JRE, and JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist.

It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent.

There are three notions of the JVM: *specification (where working of Java Virtual Machine is specified)*, *implementation(JRE)*, and run-time *instance (Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.)*.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

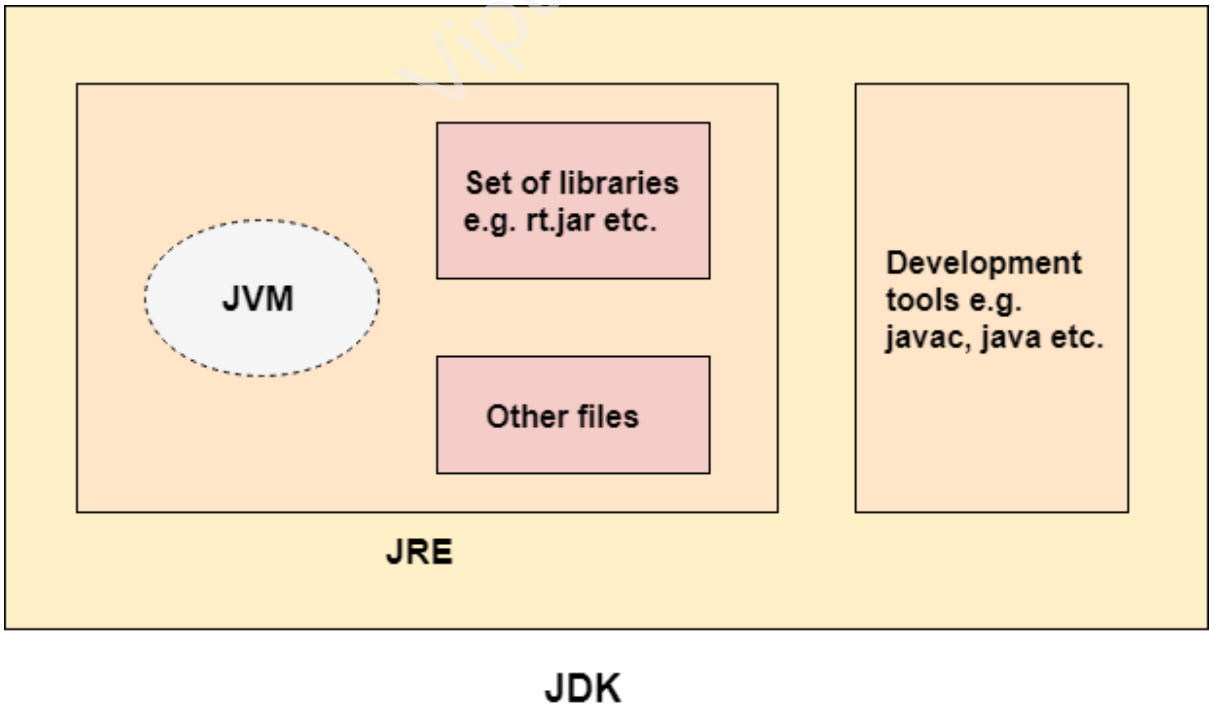
JRE is an acronym for **Java Runtime Environment**. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

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

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

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

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



JVM Architecture

It contains classloader, memory area, execution engine etc.

1) Classloader

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loads the jar files located inside the *$JAVA_HOME/jre/lib/ext* directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the class files from classpath. By default, classpath is set to the current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

These are the internal classloaders provided by Java. If you want to create your own classloader, you need to extend the ClassLoader class.

2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap

It is the runtime data area in which objects are allocated.

4) Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as the thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack

It contains all the native methods used in the application.

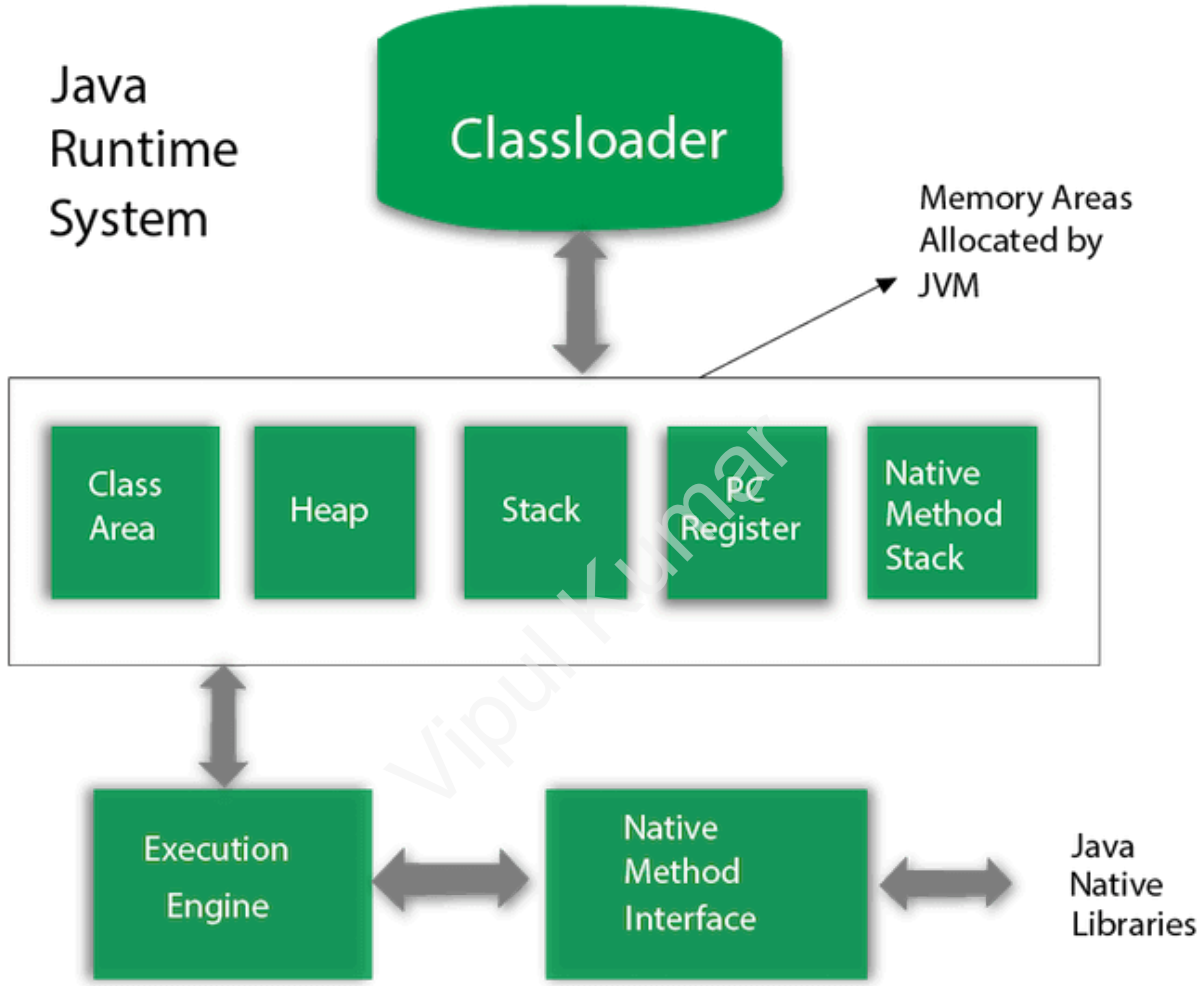
7) Execution Engine

It contains:

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

8) Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.



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.

1) Local Variable:- A variable declared inside the body of the method is called local variable. A local variable cannot be defined with the "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.

3) Static variable:- A variable that is declared as static is called a static variable. It cannot be local. We can create a single copy of the static variable and share it among all the instances of the class.

DataTypes in Java

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

1. Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

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

2. Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

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:

Operator Type	Category	Precedence
Unary	postfix	*Expr++ expr--*
	prefix	*++expr --expr +expr -expr* ~ !

Arithmetic	multiplicative	`* / %`
	additive	`+ -`
Shift	shift	`<< >> >>>`
Relational	comparison	`< > <= >= instanceof`
	equality	`== !=`
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||

Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name. There are 48 keywords in Java.

Java Control Statements

Java provides three types of control flow statements.

1. Decision Making statements
    - o if statements
    - o switch statement

2. Loop statements
    - o do while loop
    - o while loop
    - o for loop
    - o for-each loop

3. Jump statements
    - o break statement
    - o continue statement

Decision-Making statements

Decision-making statements decide which statement to execute and when.

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.

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.

3) if-else-if ladder

The if-else-if statement contains the if-statement followed by multiple elseif statements. In other words, we can say that it is the chain of if-else 10 | P a g e statements that create a decision tree where the program may enter in the block of code where the condition is true.

4. Nested if-statement

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

Switch Statements

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

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.

1. for loop

In Java, the for loop is similar to C and C++. It enables us to initialise 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.

2. 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.

3. do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iterations is not known and we have to execute the loop at least once, we can use a do-while loop. It is also known as the exit-controlled loop since the condition is not checked in advance.

3. 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.

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

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program.

Java break statement

The break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement.

Java continue statement

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

2. OOPs Concepts

Java OOPs Concepts

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

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

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.An object has three characteristics:

- **State:** It represents the data (value) of an object.
- **Behaviour:** It represents the behaviour (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** It is used internally by the JVM to identify each object uniquely.

Class

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

A class in Java can contain:

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

Instance Variable

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

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

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

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

Anonymous object

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

For example:

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

Methods in Java

A method is like a function which is used to expose the behaviour of an object. The method is a collection of instructions that performs a specific task.

Advantage of Method

- Code Reusability
- Code Optimization

Constructors in Java

A constructor is a block of codes similar to the method. It is called when an instance of the class is created. It is called a constructor because it constructs the values at the time of object creation. There are three 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 synchronised.

We can use access modifiers while declaring a constructor. There are two types of constructors in Java:

1. **Default constructor (no-arg constructor) :-** A constructor is called "Default Constructor" when it doesn't have any parameter.

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

2. **Parameterized constructor :-** A constructor which has a specific number of parameters is called a parameterized constructor. The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Constructor Overloading in Java

Constructor overloading 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.

Difference between Constructor and Method

Java Constructor	Java Method
A constructor is used to initialise the state of an object.	A method is used to expose the behaviour of an object.
A constructor must not have a return type.	A method must have a return type.

The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be the same as the class name.	The method name may or may not be the same as the class name.

Copy Constructor

There is no copy constructor in Java. There are many ways to copy the values of one object into another in Java. They are:

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

Does the constructor return any value?

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

Can constructor perform other tasks instead of initialization?

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

Is there a Constructor class in Java?

Yes.

What is the purpose of Constructor class?

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

Inheritance

When one object acquires all the properties and behaviours of a parent object, it is known as inheritance. Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship. It provides code reusability. It is used to achieve runtime polymorphism (Method Overriding).

Types of inheritance in java

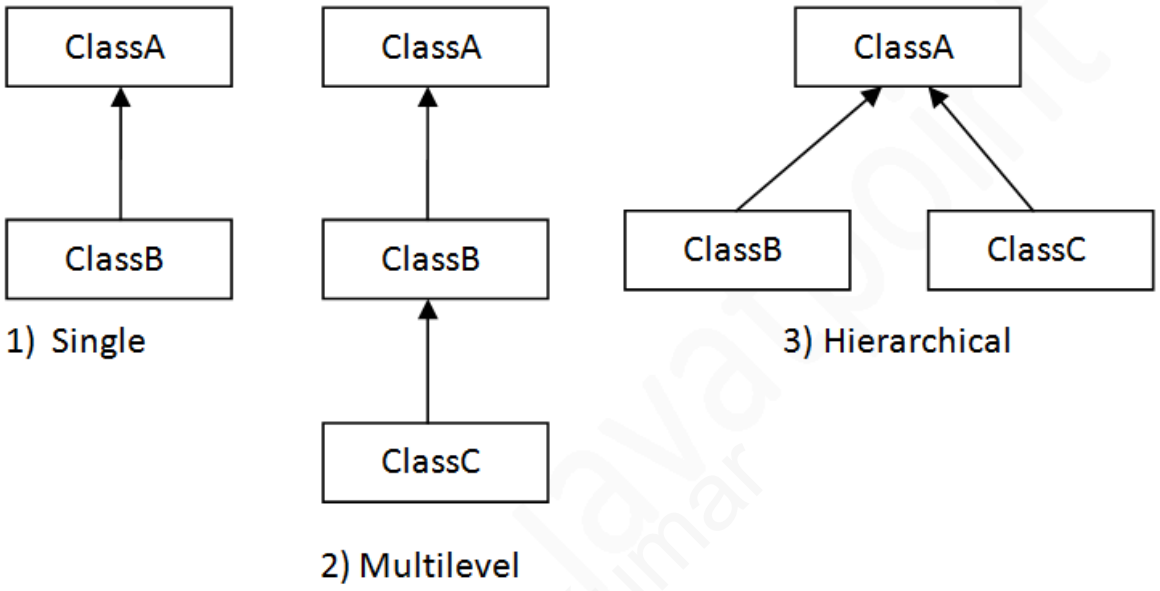
On the basis of class, there can be three types of inheritance in java: **single**, **multilevel** and **hierarchical**. In java programming, **multiple** and **hybrid** inheritance is supported through interface only.

Multiple and hybrid inheritance is not supported through class in Java.

Why is multiple inheritance not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java. Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from a child class object, there will be ambiguity when calling the method of A or B class.

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



Polymorphism

If *one task is performed in different ways*, it is known as polymorphism.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. In Java, we use **method overloading** and **method overriding** to achieve polymorphism.

If the reference variable of Parent class refers to the object of Child class, it is known as **upcasting**.

Method overloading is a **compile-time polymorphism**.

Let us create two classes Bike and Splendor. Splendour class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of the 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 the compiler, it is known as **runtime polymorphism**.

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

Method Overloading in Java

If a class has multiple methods having the same name but different in parameters, it is known as **Method Overloading**. Method overloading *increases the readability of the program*. There are two ways to overload the method in java:-

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

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

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

In java, method overloading is not possible by changing the return type of the method only because of ambiguity.

Can we overload java main() method?

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

Method Overloading with TypePromotion

```
class OverloadingCalculation1{

    void sum(int a,long b){System.out.println(a+b);}
```

```
    void sum(int a,int b,int c){System.out.println(a+b+c);}

    public static void main(String args[]){

        OverloadingCalculation1 obj=new OverloadingCalculation1();

        obj.sum(20,20);//now second int literal will be promoted to long

        obj.sum(20,20,20);

    }

}
```

Method Overriding in Java

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

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

Method overriding is used for runtime polymorphism.

Can we override the static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why can we not override the static method?

It is because the static method is bound with class whereas the instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Abstraction

A process of hiding the implementation details and showing only functionality to the user, is known as **abstraction**. For example phone calls, we don't know the internal processing. In Java, we use abstract classes and interfaces to achieve abstraction. Abstraction lets you focus on what the object does instead of how it does it.

There are two ways to achieve abstraction in java

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

Abstract Class

A class which is declared as abstract is known as an abstract class.

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

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

Interface

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

It is used to achieve abstraction and multiple inheritance in java.

Java Interface also represents the **IS-A** relationship. It cannot be instantiated just like the abstract class.

We can have **default and static methods** in an interface.

We can have **private methods** in an interface.

It can be used to achieve loose coupling.

Interface fields are public, static and final by default, and the methods are public and abstract.

A class extends another class, an interface extends another interface, but a class implements an interface.

Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

Multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

What is a marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, Serializable, Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

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 an example of encapsulation. **Java bean** is the fully encapsulated class because all the data members are private here and we use setter and getter methods to set and get the data in it.

Java Package

A java package is a group of similar types of classes, interfaces and sub-packages. Packages in java can be categorised in two forms, 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 categorise the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collisions.

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

The access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class.

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.

Coupling

Coupling refers to the knowledge or information or dependency of another class. If a class has the detailed information of another class, there is strong coupling. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts.

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Aggregation

Aggregation is a way to achieve Association. It represents the weak relationship between objects. It is also termed as a ***has-a*** relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

For example, an Employee has an object of Address, address object contains its own information such as city, state, country etc. In such a case the relationship is Employee **HAS-A** address.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object.

Java Naming Convention

Identifiers Type	Naming Rules	Examples

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

Interface	It should start with the uppercase letter.
	It should be an adjective such as Runnable, Remote, ActionListener.
	Use appropriate words, instead of acronyms.
		interface **Printable**
		{
		//code snippet
		}

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

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

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

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

static keyword

The **static keyword** in Java is used for memory management mainly. The static keyword belongs to the class rather 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

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.

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

Static Method

If you apply a static keyword with any method, it is known as a 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 a static data member and can change the value of it.

Restrictions for the static method

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

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

Why is the Java main method static?

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

Static Block

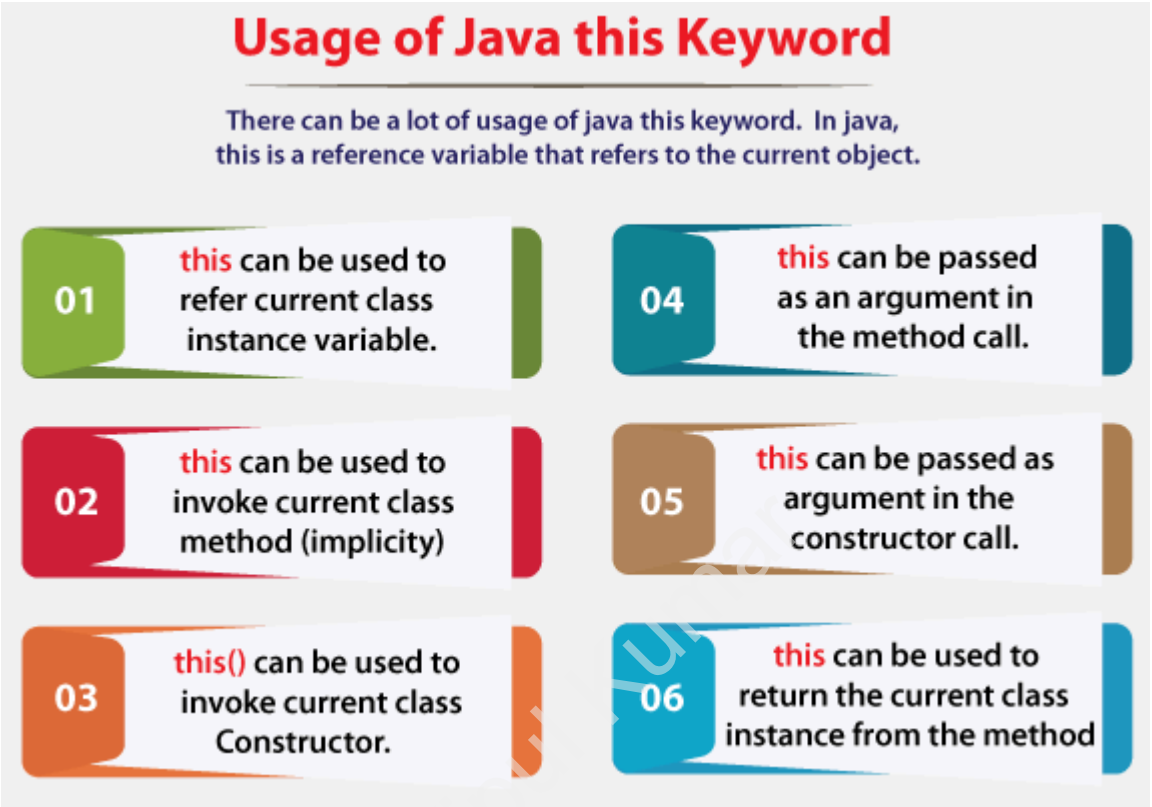
- Is used to initialise the static data member.
- It is executed before the main method at the time of classloading.

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

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

this keyword

In Java, this is a **reference variable** that refers to the current object.



super keyword

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

Usage of Java super Keyword

1. super can be used to refer to an immediate parent class instance variable. (**super**.color)
2. super can be used to invoke the immediate parent class method. (**super**.eat();)
3. super() can be used to invoke immediate parent class constructor.

final keyword

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

1. variable
2. method
3. class

final variable

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

final method

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

Final class

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

Is the final method inherited?

Yes, the final method is inherited but you cannot override it.

What is the blank or uninitialized final variable?

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

static blank final variable

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

Can we declare a constructor final?

No, because the constructor is never inherited.

What is the final parameter?

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

Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass. Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override a method by changing the return type if a subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type.

Instance Initializer block

It is used to initialise the instance data member. It runs each time an object of the class is created.

1. The instance initializer block is created when an instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding)

When the type of the object is determined at compile time(by the compiler), it is known as static binding. If there is any private, final or static method in a class, there is static binding. **Dog d1=new Dog();**

2. Dynamic Binding (also known as Late Binding).

When the type of the object is determined at run-time, it is known as dynamic binding. **Animal a=new Dog();**

Java OOPs Miscellaneous

The **java instanceof operator** is used to test whether the object is an instance of the specified type.

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java. The Object class is beneficial if you want to refer to any object whose type you don't know.

Object **cloning** is a way to create an exact copy of an object. The clone() method of Object class is used to clone an object.

Java **Math** class provides several methods to work on maths calculations like Math.min(), Math.max(), Math.avg(), Math.sin(), Math.cos(), Math.tan(), Math.round(), Math.ceil(), Math.floor(), Math.abs() etc.

There is only **call by value** in java, not **call by reference**.

Java **strictfp keyword** ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform, that is why java programming language has provided the strictfp keyword, so that you get the same result on every platform. So, now you have better control over the floating-point arithmetic.

Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*. Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronisation, etc.
The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

The automatic conversion of primitive data type into its corresponding wrapper class is known as **autoboxing**, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

The automatic conversion of wrapper type into its corresponding primitive type is known as **unboxing**. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Java Array

Java array is an object which contains elements of a similar data type.

Syntax to Declare an Array in Java

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

Instantiation of an Array in Java

```
arrayRefVar=new data type[size];
```

Syntax to Declare Multidimensional Array in Java

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

Example to instantiate Multidimensional Array in Java

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

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

```
for(data_type variable:array){
        //body of the loop
}
```


3. String

String is an object that represents a sequence of characters. An array of characters works the same as Java string. The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.

The CharSequence interface is used to represent the sequence of characters. **String**, **StringBuffer** and **StringBuilder** classes implement it. It means, we can create strings in Java by using these three classes.

The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

There are two ways to create String object:

1. By string literal

Java String literal is created by using double quotes. Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

a. `String s1="Welcome";`

b. `String s2="Welcome"; //It doesn't create a new instance`

Why does Java use the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2. By new keyword

```
String s=new String("Welcome"); //creates two objects and one reference variable
```

In such a case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns a formatted string with a given locale.
5	String substring(int beginIndex)	It returns a substring for a given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.

7	**boolean contains(CharSequence s)**	**It returns true or false after matching the sequence of char values.**
8	**static String join(CharSequence delimiter, CharSequence... elements)**	**It returns a joined string.**
9	**static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)**	**It returns a joined string.**
10	**boolean equals(Object another)**	**It checks the equality of string with the given object.**
11	**boolean isEmpty()**	**It checks if the string is empty.**
12	**String concat(String str)**	**It concatenates the specified string.**
13	**String replace(char old, char new)**	**It replaces all occurrences of the specified char value.**

14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check the case.
16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.
18	String intern()	It returns an interned string.
19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with the given index.

21	**int indexOf(String substring)**	**It returns the specified substring index.**
22	**int indexOf(String substring, int fromIndex)**	**It returns the specified substring index starting with the given index.**
23	**String toLowerCase()**	**It returns a string in lowercase.**
24	**String toLowerCase(Locale l)**	**It returns a string in lowercase using the specified locale.**
25	**String toUpperCase()**	**It returns a string in uppercase.**
26	**String toUpperCase(Locale l)**	**It returns a string in uppercase using the specified locale.**

27	String trim()	It removes the beginning and ending spaces of this string.
28	static String valueOf(int value)	It converts a given type into a string. It is an overloaded method.

In Java, String objects are immutable. Immutable simply means unmodifiable or unchangeable.

Once a String object is created its data or state can't be changed but a new String object is created.

Why are String objects immutable in Java?

Java uses the concept of String literal. Suppose there are 5 reference variables, all referring to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Why is the String class is Final in Java?

The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones.

String Comparison

String can be compared on the basis of content and reference.

It is used in authentication (by equals() method), sorting (by compareTo() method), reference matching (by == operator) etc.

There are three ways to compare String in Java:

1. By Using equals() Method

The String class equals() method compares the original content of the string. It compares values of string for equality.

2. By Using == Operator

The == operator compares references not values.

3. By compareTo() Method

The String class compareTo() method compares values lexicographically and returns an integer value that describes if the first string is less than, equal to or greater than the second string.

Suppose s1 and s2 are two String objects. If:

- s1 == s2 : The method returns 0.
- s1 > s2 : The method returns a positive value.
- s1 < s2 : The method returns a negative value.

String Concatenation

String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

1. **By + (String concatenation) operator**
2. **By concat() method**

Substring in Java

A part of String is called substring. In other words, a substring is a subset of another String. substring from the given String object by one of the two methods:

1. **public String substring(int startIndex)**
2. **public String substring(int startIndex, int endIndex) :-startIndex: inclusive, endIndex: exclusive**

StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. It is mutable i.e. it can be changed. Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

- **The *append()* method concatenates the given argument with this String.**
- **The *insert()* method inserts the given String with this string at the given position.**
- **The *replace()* method replaces the given String from the specified beginIndex and endIndex.**
- **The *delete()* method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.**
- **The *reverse()* method of the StringBuilder class reverses the current String.**
- **The *capacity()* method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases**

the capacity by (old capacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

- **The *ensureCapacity()* method of the StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (old capacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.**

StringBuilder Class

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is the same as the StringBuffer class except that it is non-synchronized.

- **The StringBuilder *append()* method concatenates the given argument with this String.**
- **The StringBuilder *insert()* method inserts the given string with this string at the given position.**
- **The StringBuilder *replace()* method replaces the given string from the specified beginIndex and endIndex.**
- **The *delete()* method of StringBuilder class deletes the string from the specified beginIndex to endIndex.**
- **The *reverse()* method of StringBuilder class reverses the current string.**
- **The *capacity()* method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (old capacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.**
- **The *ensureCapacity()* method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (old capacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.**

Difference between String and StringBuffer

1	The String class is immutable.	The StringBuffer class is mutable.
2	String is slow and consumes more memory when we concatenate too many strings because every time it creates a new instance.	StringBuffer is fast and consumes less memory when we concatenate the strings.
3	String class overrides the equals() method of Object class. So you can compare the contents of two strings by the equals() method.	The StringBuffer class doesn't override the equals() method of Object class.
4	String class is slower while performing concatenation operation.	The StringBuffer class is faster while performing concatenation operations.
5	String class uses String constant pool.	StringBuffer uses Heap memory

Difference between StringBuffer and StringBuilder

1	StringBuffer is *synchronised* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.

2	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.
3	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

How to create an Immutable class?

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String classes are immutable. We can also create an immutable class by creating a final class that has final data members.

- The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
- The class is final so we cannot create the subclass.
- There are no setter methods i.e. we have no option to change the value of the instance variable.

These points make the class immutable.

Java toString() Method

To represent any object as a string, the **toString()** method comes into existence.

The **toString()** method returns the String representation of the object.

If you print any object, Java compiler internally invokes the **toString()** method on the object. So overriding the **toString()** method, returns the desired output, it can be the state of an object etc. depending on your implementation.

4. Regular Expression

The Java Regex or Regular Expression is an API to *define a pattern for searching or manipulating strings*. The java.util.regex package provides the following classes and interfaces for regular expressions.

1. **MatchResult interface :-**
2. **Matcher class :-** **It implements the MatchResult interface. It is a *regex engine* which is used to perform match operations on a character sequence.**
3. **Pattern class :-** **It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.**
4. **PatternSyntaxException class**

There are three ways to write the regex example in Java.

```
1. import java.util.regex.*;
2. public class RegexExample1{
3. public static void main(String args[]){
4. //1st way
5. Pattern p = Pattern.compile(".s");//. represents single character
6. Matcher m = p.matcher("as");
7. boolean b = m.matches();
8.
9. //2nd way
10.boolean b2=Pattern.compile(".s").matcher("as").matches();
11.
12.//3rd way
13.boolean b3 = Pattern.matches(".s", "as");
14.
15.System.out.println(b+" "+b2+" "+b3);
16.}}
```


Regex Character classes

No.	Character Class	Description
1	[abc]	a, b, or c (simple class)
2	[^abc]	Any character except a, b, or c (negation)
3	[a-zA-Z]	a through z or A through Z, inclusive (range)
4	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
5	[a-z&&[def]]	d, e, or f (intersection)
6	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
7	[a-z&&[^m-p]]	a through z, and not m through p: [a-la-z](subtraction)

Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

Regex	Description

X?	X occurs once or not at all
X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only
X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

Regex Metacharacters

The regular expression metacharacters work as shortcodes.

Regex	Description
.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]

\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]
\b	A word boundary
\B	A non word boundary

5. Exception Handling

Exception Handling in Java

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

What is Exception in Java?

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?

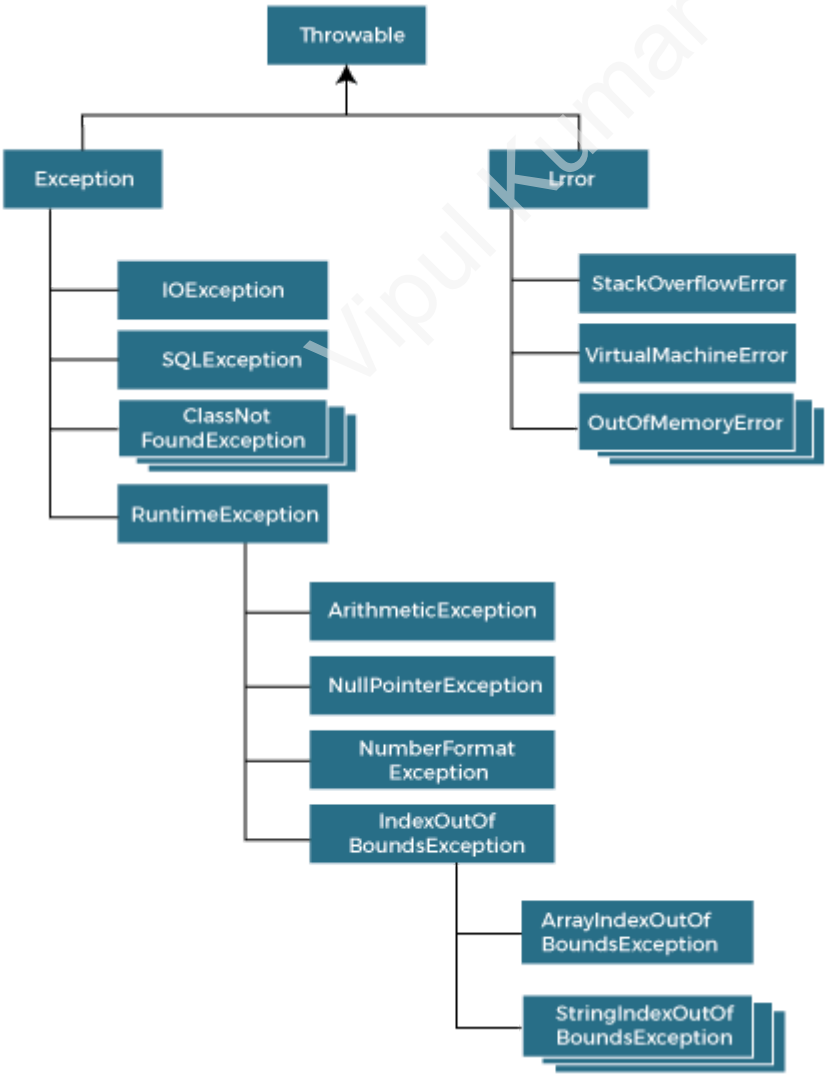
Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

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.

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.



Types of Java Exceptions

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

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

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

2) Unchecked Exception

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

3) Error

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

Java Exception Keywords

Java provides five keywords that are used to handle the exception.

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 a 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 signatures.

try-catch 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 a try block that will not throw an exception.**
- **Java try block must be followed by either catch or finally block.**
- **Java catch block is used to handle the Exception by declaring the type of exception within the parameter.**
- **The catch block must be used after the try block only.**
- **We can use multiple catch blocks with a single try block.**

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., the rest of the code is executed.

Multi-catch block

- **A try block can be followed by one or more catch blocks.**
- **Each catch block must contain a different exception handler.**
- **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.**
- **In case, a specific type exception occurs and doesn't provide the corresponding exception type. In such a case, the catch block containing the parent exception class Exception will be invoked.**
- **If we don't maintain order of exception declaration in the catch block (i.e. from most specific to most general), then there will be a compile time error.**

Nested try block

In Java, using a try block inside another try block is permitted. It is called a nested try block.

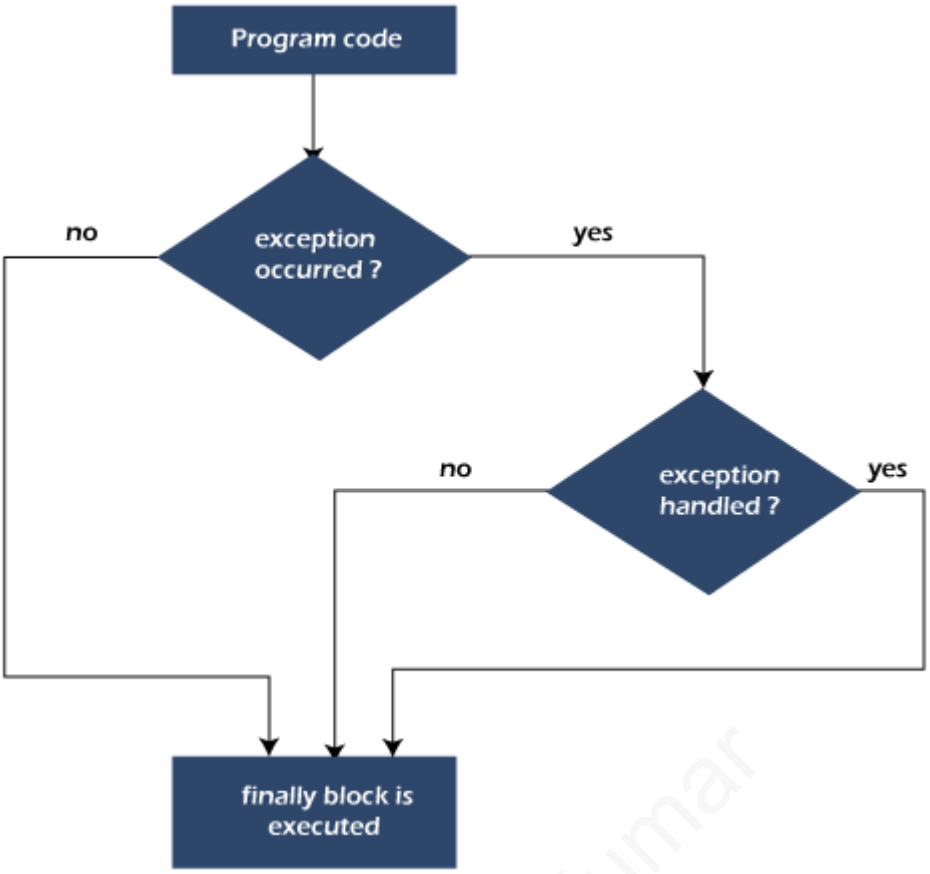
Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

- **When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block is checked for that exception, and if it matches, the catch block of the outer try block is executed.**
- **If none of the catch blocks specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.**

finally block

- **Java finally block is a block used to execute important code such as closing the connection, etc.**
- **If we don't handle the exception, before terminating the program, JVM executes the 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.**
- **For each try block there can be zero or more catch blocks, but only one finally block.**
- **The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).**



throw keyword

- The Java throw keyword is used to throw an exception explicitly.
- We specify the exception object which is to be thrown.
- The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either checked or unchecked exceptions in Java with the throw keyword. It is mainly used to throw a custom exception.
- We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw an ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw an exception using throw keyword.
- The syntax of the Java throw keyword is given below.

  throw Instance i.e.,

```
throw new IOException("sorry device error");
```

- The Instance must be of type Throwable or subclass of Throwable. For example, Exception is the subclass of Throwable and the user-defined exceptions usually extend the Exception class.
- If we throw a checked exception using throw keyword, it must handle the exception using catch block or the method must declare it using throws declaration.
- Every subclass of Error and RuntimeException is an unchecked exception in Java.
- A checked exception is everything else under the Throwable class.

Exception Propagation

- An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

- **By default Unchecked Exceptions are forwarded in the calling chain (propagated).**

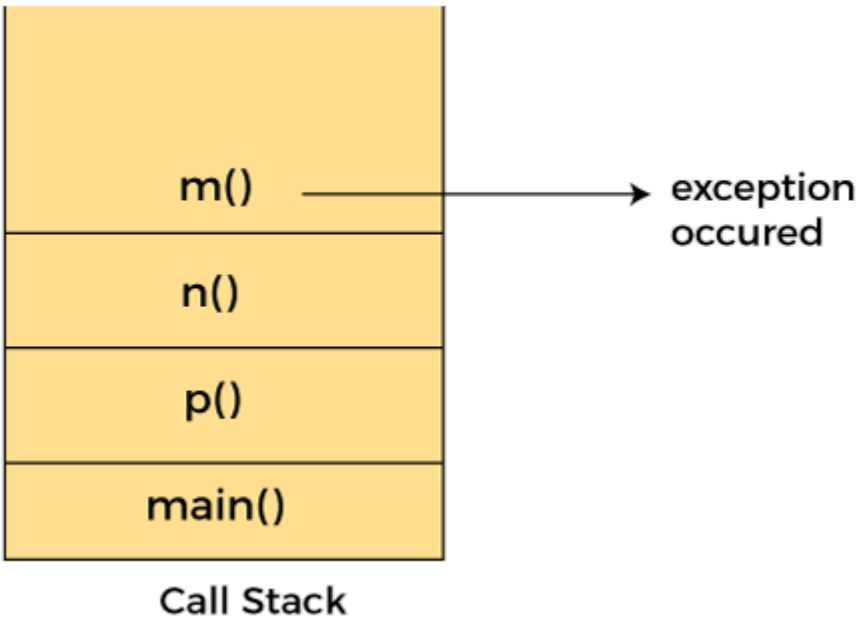

```
class TestExceptionPropagation1{
 void m(){
  int data=50/0;
 }
 void n(){
  m();
 }
 void p(){
  try{
   n();
  }catch(Exception e){
```

```
    System.out.println("exception handled");
    }
  }
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output

```
exception handled
      normal flow...
```

- In the above example an exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.
- Exceptions can be handled in any method in the call stack either in the main() method, p() method, n() method or m() method.



- **By default, Checked Exceptions are not forwarded in the calling chain (propagated).**
- **It is forwarded after using throws declaration.(Self said after practising)**

throws keyword

- The Java throws keyword is used to declare an exception.
- Exception Handling is mainly used to handle the checked exceptions.
- If there occurs any unchecked exception such as NullPointerException, it is the programmers' fault that he is not checking the code before it is being used.
- Syntax :

```
return_type method_name() throws exception_class_name{
        //method code
}
```

Which exception should be declared?

Ans: Checked exception only, because:

- **unchecked exception: under our control so we can correct our code.**
- **error: beyond our control. For example, we are unable to do anything if VirtualMachineError or StackOverflowError occurs.**

Advantage of Java throws keyword

- **Now Checked Exceptions can be propagated (forwarded in call stack).**
- **It provides information to the caller of the method about the exception.**

If we are calling a method that declares an exception, we must either catch or declare the exception.

There are two cases:

Case 1: We have caught the exception i.e. we have handled the exception using try/catch block.

Case 2: **We have declared the exception i.e. specified throws keyword with the method.**

- **In case we declare an exception, if an exception does not occur, the code will be executed fine.**
- **In case we declare the exception and the exception occurs, it will be thrown at runtime because throws does not handle the exception. Rest of the code will not execute.**

Que) Can we rethrow an exception?

Yes, by throwing the same exception in the catch block.

Difference between throw and throws keyword

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used to 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 exceptions 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.

5.	**Internal implementation**	**We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.**	**We can declare multiple exceptions using throws keywords that can be thrown by the method. For example, main() throws IOException, SQLException.**

Difference between final, finally and finalize keyword

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 an 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, the final variable becomes constant and cannot be modified. (2) final method cannot be overridden by subclass. (3) final class cannot be inherited.	(1) finally block runs the important code even if an 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 the block is executed as soon as the try-catch block is executed.	The finalize method is executed just before the object is destroyed.

			Its execution is not dependent on the exception.	

Exception Handling with Method Overriding

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

These rules are listed below:

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

Custom Exception

Why use custom exceptions?

Java exceptions cover almost all the general types of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

- **To catch and provide specific treatment to a subset of existing Java exceptions.**

- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exceptions, we need to extend the Exception class that belongs to the java.lang package.

We need to write the constructor that takes the String as the error message and it is called parent class constructor.

6. Inner Classes

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

Advantage of Java inner classes

There are three advantages of inner classes in Java. They are as follows:

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 groups 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**
- **Static nested class**

Member inner class

A non-static class that is created inside a class but outside a method is called member inner class. It is also known as a regular inner class. It can be declared with access modifiers like public, default, private, and protected.

Syntax to instantiate member inner class:

```
OuterClassReference.new MemberInnerClassConstructor();
```

Internal working of Java member inner class

The java compiler creates two class files in the case of the inner class. The class file name of the inner class is "Outer$Inner". If you want to instantiate the inner class, you must have to create an instance of the outer class. In such a case, an instance of the inner class is created inside the instance of the outer class.

Anonymous Inner class

Java anonymous inner class is an inner class without a name and for which only a single object is created.

It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. **Class (may be abstract or concrete).**

2. Interface

Internal working of given code

```
Person p=new Person(){

    void eat(){System.out.println("nice fruits");}

};
```

1. A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.
2. An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

Internal working of given code

It performs two main tasks behind this code:

```
Eatable p=new Eatable(){

    void eat(){System.out.println("nice fruits");}

};
```

1. A class is created, but its name is decided by the compiler, which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of the Anonymous class is created that is referred to by 'p', a reference variable of the Eatable type.

The compiler creates a class named Simple$1 that has the reference of the outer class.

Local Inner class

A class i.e., created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block.

Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause.

Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them.

However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Local variables can't be private, public, or protected.

Local inner class cannot be invoked from outside the method.

Local inner class cannot access non-final local variables till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in the local inner class.

The compiler creates a class named Simple$1Local that has the reference of the outer class.

Static nested class

A static class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- **It can access static data members of the outer class, including private.**

- The static nested class cannot access non-static (instance) data members.

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

In this example, you need to create the instance of a static nested class because it has an instance method msg(). But you don't need to create the object of the Outer class because the nested class is static and static properties, methods, or classes can be accessed without an object.

If you have the static member inside the static nested class, you don't need to create an instance of the static nested class.

Nested Interface

- An interface, i.e., declared within another interface or class, is known as a nested interface.
- The nested interface must be referred to by the outer interface or class. It can't be accessed directly.
- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static.

- The java compiler internally creates a public and static interface.

Can we define a class inside the interface?

Yes, if we define a class inside the interface, the Java compiler creates a static nested class. Let's see how can we define a class within the interface:

```
interface M{
    class A{
    }
}
```

7. Multithreading

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading rather than multiprocessing because threads use a shared memory area. They don't allocate a separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilise the CPU. Multitasking can be achieved in two ways:

- **Process-based Multitasking (Multiprocessing)**
- **Thread-based Multitasking (Multithreading)**

1) Process-based Multitasking (Multiprocessing)

- **Each process has an address in memory. In other words, each process allocates a separate memory area.**
- **The process is heavyweight.**
- **Cost of communication between the processes is high.**
- **Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.**

2) Thread-based Multitasking (Multithreading)

- **Threads share the same address space.**
- **A thread is lightweight.**
- **Cost of communication between the threads is low.**

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there exception occurs in one thread, it doesn't affect other threads. It uses a shared memory area.

A thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

Java Thread class

Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. **New**
2. **Active**
3. **Blocked / Waiting**
4. **Timed Waiting**
5. **Terminated**

New: **Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.**

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is runnable, and the other is running.

Runnable: A thread that is ready to run is then moved to the runnable state. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread to the running state. All those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

Running: When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

Timed Waiting: A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and starts its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

When a thread has finished its job, then it exists or terminates normally.

Abnormal termination: It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

One can get the current state of a thread using the Thread.getState() method. The java.lang.Thread.State class of Java provides the constants ENUM to represent the state of a thread. These constants are:

1. **public static final Thread.State NEW**
2. **public static final Thread.State RUNNABLE**
3. **public static final Thread.State BLOCKED**
4. **public static final Thread.State WAITING**
5. **public static final Thread.State TIMED_WAITING**
6. **public static final Thread.State TERMINATED**

How to create a thread in Java

There are two ways to create a thread:

1. **By extending Thread class**
2. **By implementing Runnable interface**

Thread class

Thread class provides constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Constructors of Thread Class

- **Thread()**
- **Thread(String name)**
- **Thread(Runnable r)**

- **Thread(Runnable r,String name)**

Methods of Thread class

1. **public void run(): is used to perform action for a thread.**
2. **public void start(): starts the execution of the thread.JVM calls the run() method on the thread.**
3. **public void sleep(long milliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.**
4. **public void join(): waits for a thread to die.**
5. **public void join(long milliseconds): waits for a thread to die for the specified milliseconds.**
6. **public int getPriority(): returns the priority of the thread.**
7. **public int setPriority(int priority): changes the priority of the thread.**
8. **public String getName(): returns the name of the thread.**
9. **public void setName(String name): changes the name of the thread.**
10. **public Thread currentThread(): returns the reference of currently executing thread.**
11. **public int getId(): returns the id of the thread.**
12. **public Thread.State getState(): returns the state of the thread.**
13. **public boolean isAlive(): tests if the thread is alive.**
14. **public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.**
15. **public void suspend(): is used to suspend the thread(deprecated).**
16. **public void resume(): is used to resume the suspended thread(deprecated).**
17. **public void stop(): is used to stop the thread(deprecated).**
18. **public boolean isDaemon(): tests if the thread is a daemon thread.**

19. **public void setDaemon(boolean b): marks the thread as daemon or user thread.**
20. **public void interrupt(): interrupts the thread.**
21. **public boolean isInterrupted(): tests if the thread has been interrupted.**
22. **public static boolean interrupted(): tests if the current thread has been interrupted.**

Runnable interface

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interfaces have only one method named run().

1. **public void run(): is used to perform action for a thread.**

Starting a thread

The start() method of Thread class is used to start a newly created thread. It performs the following tasks:

- **A new thread starts(with a new call stack).**
- **The thread moves from the New state to the Runnable state.**
- **When the thread gets a chance to execute, its target run() method will run.**

Thread Scheduler in Java

A component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler in Java.

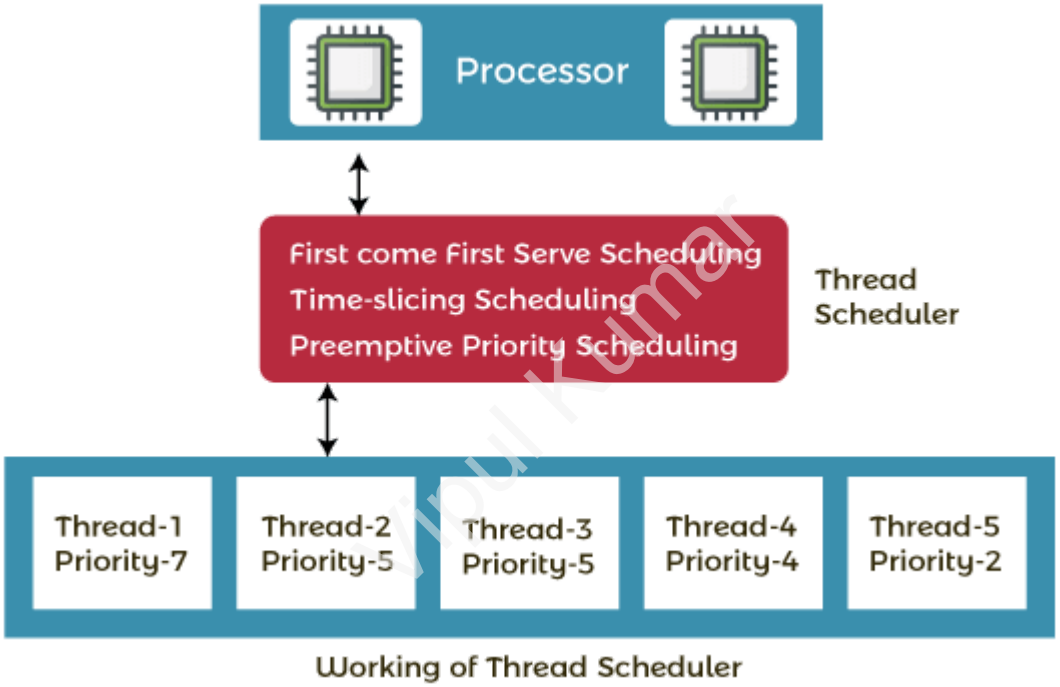
In Java, a thread is only chosen by a thread scheduler if it is in the runnable state.

However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. Priority and Time of arrival.

Priority: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

Time of Arrival: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, the arrival time of the thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Working of the Java Thread Scheduler



Working of Thread Scheduler

It is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is preempted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) have the same priorities and arrival time, the scheduling will be decided on the basis of the FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

Thread.sleep() in Java

The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

Following are the syntax of the sleep() method.

1. `public static void sleep(long mls) throws InterruptedException`
2. `public static void sleep(long mls, int n) throws InterruptedException`

The method sleep() with one parameter is the native method, and the implementation of the native method is accomplished in another programming language.

The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword.

The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

The following are the parameters used in the sleep() method.

mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

Can we start a thread twice

No. After starting a thread, it can never be started again. If you do so, an *IllegalThreadStateException* is thrown. In such a case, the thread will run once but for a second time, it will throw an exception.

What if we call the Java run() method directly instead of the start() method?

- **Each thread starts in a separate call stack when we call start() method.**
- **Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.**
- **There is no context-switching between threads because here t1 and t2 will be treated as normal objects not thread objects.**

Java join() method

The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread finishes its execution.

There are three overloaded join() methods.

join(): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the InterruptedException.

Syntax:

```
public final void join() throws InterruptedException
```

join(long mls): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked is dead or the wait for the specified time frame(in milliseconds) is over.

Syntax:

```
public final synchronized void join(long mls) throws InterruptedException, where mls is in milliseconds
```

join(long mls, int nanos): When the join() method is invoked, the current thread stops its execution and goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked is dead or the wait for the specified time frame(in milliseconds + nanos) is over.

Syntax:

```
public final synchronized void join(long mls, int nanos) throws InterruptedException
```


Naming Thread and Current Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name, i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using the setName() method. The syntax of setName() and getName() methods are given below:

1. public String getName(): is used to return the name of a thread.
2. public void setName(String name): is used to change the name of a thread.

We can also set the name of a thread directly when we create a new thread using the constructor of the class.

The currentThread() method returns a reference of the currently executing thread.

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10.

Java programmers can assign the priorities of a thread explicitly in a Java program.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assigns the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. **public static int MIN_PRIORITY**
2. **public static int NORM_PRIORITY**
3. **public static int MAX_PRIORITY**

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

A thread with high priority will get preference over lower priority threads when it comes to the execution of threads.

If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

If the value of the parameter *newPriority* of the method setPriority() goes out of the range (1 to 10), then we get the IllegalArgumentException.

If we set the priority of the main thread to any value(say 7), then the thread which is the child of the main thread will have the same priority.

Daemon Thread in Java

Daemon thread in Java is a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.

Its life depends on user threads.

It is a low priority thread.

Why does the JVM terminate the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to the user thread for background supporting tasks. If there is no user thread, why should JVM keep running this thread? That is why JVM terminates the daemon thread if there is no user thread.

The java.lang.Thread class provides two methods for java daemon thread.

1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

Java Thread Pool

Thread pool represents a group of worker threads that are waiting for the job and reused many times.

In the case of a thread pool, a group of fixed-size threads is created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, the thread is contained in the thread pool again.

newFixedThreadPool(int s): This method creates a thread pool of the fixed size s.

newCachedThreadPool(): This method creates a new thread pool that creates the new threads when needed but will still use the previously created thread whenever they are available to use.

newSingleThreadExecutor(): This method creates a new thread.

Advantage of Java Thread Pool

Better performance: It saves time because there is no need to create a new thread.

Real time usage

It is used in Servlet and JSP where the container creates a thread pool to process the request.

```
1.  public class TestThreadPool {
2.      public static void main(String[] args) {
3.        ExecutorService executor = Executors.newFixedThreadPool(5);//creating
      a pool of 5 threads
4.        for (int i = 0; i < 10; i++) {
5.          Runnable worker = new WorkerThread("" + i);
6.          executor.execute(worker);//calling execute method of ExecutorService
7.         }
8.        executor.shutdown();
9.        while (!executor.isTerminated()) {   }
10.
11.       System.out.println("Finished all threads");
12.   }
13. }
```

When one wants to execute 50 tasks but is not willing to create 50 threads. In such a case, one can create a pool of 10 threads. Thus, 10 out of 50 tasks are assigned, and the rest are put in the queue. Whenever any thread out of 10 threads becomes idle, it picks up the 11th task. The other pending tasks are treated the same way.

The following are the risk involved in the thread pools:-

Deadlock : Consider a scenario where all the threads that are executing are waiting for the results from the threads that are blocked and waiting in the queue because of the non-availability of threads for the execution.

Thread Leakage : Leakage of threads occurs when a thread is being removed from the pool to execute a task but is not returning to it after the completion of the task. For example, when a thread throws the exception and the pool class is not able to catch this exception, then the thread exits and reduces the thread pool size by 1.

Resource Thrashing : A lot of time is wasted in context switching among threads when the size of the thread pool is very large. Whenever there are more threads than the optimal number may cause the starvation problem, and it leads to resource thrashing.

Points to Remember

- Do not queue the tasks that are concurrently waiting for the results obtained from the other tasks. It may lead to a deadlock situation, as explained above.
- Care must be taken whenever threads are used for the operation that is long-lived. It may result in the waiting of thread forever and will finally lead to the leakage of the resource.
- In the end, the thread pool has to be ended explicitly. If it does not happen, then the program continues to execute, and it never ends. Invoke the shutdown() method on the thread pool to terminate the executor. Note that if someone tries to send another task to the executor after shutdown, it will throw a RejectedExecutionException.
- One needs to understand the tasks to effectively tune the thread pool. If the given tasks are contrasting, then one should look for pools for

executing different varieties of tasks so that one can properly tune them.

- To reduce the probability of running JVM out of memory, one can control the maximum threads that can run in JVM. The thread pool cannot create new threads after it has reached the maximum limit.
- A thread pool can use the same used thread if the thread has finished its execution. Thus, the time and resources used for the creation of a new thread are saved.

ThreadGroup in Java

Java thread group is implemented by *java.lang.ThreadGroup* class.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its parent thread group or any other thread groups.

There are only two constructors of the ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with a given name.

2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with a given parent group and name.

There are many methods in the ThreadGroup class. A list of ThreadGroup methods is given below.

S.N.	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	int	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.
3)	int	activeGroupCount()	This method returns an estimate of the number of active groups in the thread group and its subgroups.

4)	void	destroy()	This method destroys the thread group and all of its subgroups.
5)	int	enumerate(Thread[] list)	This method copies into the specified array every active thread in the thread group and its subgroups.
6)	int	getMaxPriority()	This method returns the maximum priority of the thread group.
7)	String	getName()	This method returns the name of the thread group.
8)	ThreadGroup	getParent()	This method returns the parent of the thread group.
9)	void	interrupt()	This method interrupts all threads in the thread group.
10)	boolean	isDaemon()	This method tests if the thread group is a daemon thread group.

11)	void	setDaemon(boolean daemon)	This method changes the daemon status of the thread group.
12)	boolean	isDestroyed()	This method tests if this thread group has been destroyed.
13)	void	list()	This method prints information about the thread group to the standard output.
14)	boolean	parentOf(ThreadGroup g	This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	void	suspend()	This method is used to suspend all threads in the thread group.
16)	void	resume()	This method is used to resume all threads in the thread group which was

			suspended using suspend() method.
17)	void	setMaxPriority(int pri)	This method sets the maximum priority of the group.
18)	void	stop()	This method is used to stop all threads in the thread group.
19)	String	toString()	This method returns a string representation of the Thread group.


```
ThreadGroup tg1 = new ThreadGroup("Group A");
Thread t1 = new Thread(tg1,new MyRunnable(),"one");
Thread t2 = new Thread(tg1,new MyRunnable(),"two");
Thread t3 = new Thread(tg1,new MyRunnable(),"three");
```

Now we can interrupt all threads by a single line of code only.

```
Thread.currentThread().getThreadGroup().interrupt();
```

Java Shutdown Hook

A special construct that facilitates the developers to add some code that has to be run when the Java Virtual Machine (JVM) is shutting down is known as the **Java shutdown hook**.

For registering the instance of the derived class as the shutdown hook, one has to invoke the method **Runtime.getRuntime().addShutdownHook(Thread)**, whereas for removing the already registered shutdown hook, one has to invoke the **removeShutdownHook(Thread)** method.

The **addShutdownHook()** method of the Runtime class is used to register the thread with the Virtual Machine.

Syntax:

```
public void addShutdownHook(Thread hook){}
```

The object of the Runtime class can be obtained by calling the static factory method getRuntime(). For example:

```
Runtime r = Runtime.getRuntime();
```

The removeShutdownHook() method of the Runtime class is invoked to remove the registration of the already registered shutdown hooks.

Syntax:

```
public boolean removeShutdownHook(Thread hook){ }
```

True value is returned by the method, when the method successfully de-register the registered hooks; otherwise returns false.

The method that returns the instance of a class is known as the **factory method**.

How to perform a single task by multiple threads in Java?

If you have to perform a single task by many threads, have only one run() method.

For example:

```
class TestMultitasking1 extends Thread{
 public void run(){
  System.out.println("task one");
 }
 public static void main(String args[]){
  TestMultitasking1 t1=new TestMultitasking1();
  TestMultitasking1 t2=new TestMultitasking1();
  TestMultitasking1 t3=new TestMultitasking1();
  t1.start();
  t2.start();
  t3.start();
 }
}
```


How to perform multiple tasks by multiple threads (multitasking in multithreading)?

If you have to perform multiple tasks by multiple threads,have multiple run() methods.

For example:

```
class Simple1 extends Thread{
 public void run(){
   System.out.println("task one");
 }
}

class Simple2 extends Thread{
 public void run(){
   System.out.println("task two");
 }
}

 class TestMultitasking3{
 public static void main(String args[]){
  Simple1 t1=new Simple1();
  Simple2 t2=new Simple2();

  t1.start();
  t2.start();
 }
}
```

Note : Each thread runs in a separate call stack.

Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is the process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy unused objects.

To do so, we were using the free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

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

How can an object be unreferenced?

There are many ways:

1) By nulling a reference

```
Employee e=new Employee();

e=null;
```

2) By assigning a reference to another

```
Employee e1=new Employee();

Employee e2=new Employee();
```

```
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object

```
new Employee();
```

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){}
```

Note: The Garbage collector of JVM collects only those objects that are created by new keywords. So if you have created any object without new, you can use the finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

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

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

Note: Neither finalization nor garbage collection is guaranteed.

Runtime class

Java Runtime class is used *to interact with the Java runtime environment*. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class available for one java application.

The Runtime.getRuntime() method returns the singleton instance of Runtime class.

Important methods of Java Runtime class

No.	Method	Description
1)	public static Runtime getRuntime()	returns the instance of Runtime class.
2)	public void exit(int status)	terminates the current virtual machine.
3)	public void addShutdownHook(Thread hook)	registers new hook thread.
4)	public Process exec(String command)throws IOException	executes given command in a separate process.
5)	public int availableProcessors()	returns no. of available processors.
6)	public long freeMemory()	returns amount of free memory in JVM.
7)	public long totalMemory()	returns amount of total memory in JVM.

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is a better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

1. **To prevent thread interference.**
2. **To prevent consistency problems.**

Types of Synchronization

There are two types of synchronization

1. **Process Synchronization**
2. **Thread Synchronization**

Thread Synchronization

There are two types of thread synchronization: mutual exclusive and inter-thread communication.

1. **Mutual Exclusive**
    1. **Synchronized method.**
    2. **Synchronized block.**
    3. **Static synchronization.**

2. **Cooperation (Inter-thread communication in java)**

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. **By Using Synchronized Method**
2. **By Using Synchronized Block**
3. **By Using Static Synchronization**

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

When there is no synchronization, then output will be inconsistent.

Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Synchronized Block

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use a synchronized block.
- If we put all the codes of the method in the synchronized block, it will work the same as the synchronized method.
- Synchronized block is used to lock an object for any shared resource.
- Scope of the synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
- The system performance may degrade because of the slower working of synchronized keywords.
- Java synchronized block is more efficient than Java synchronized method.

Syntax:

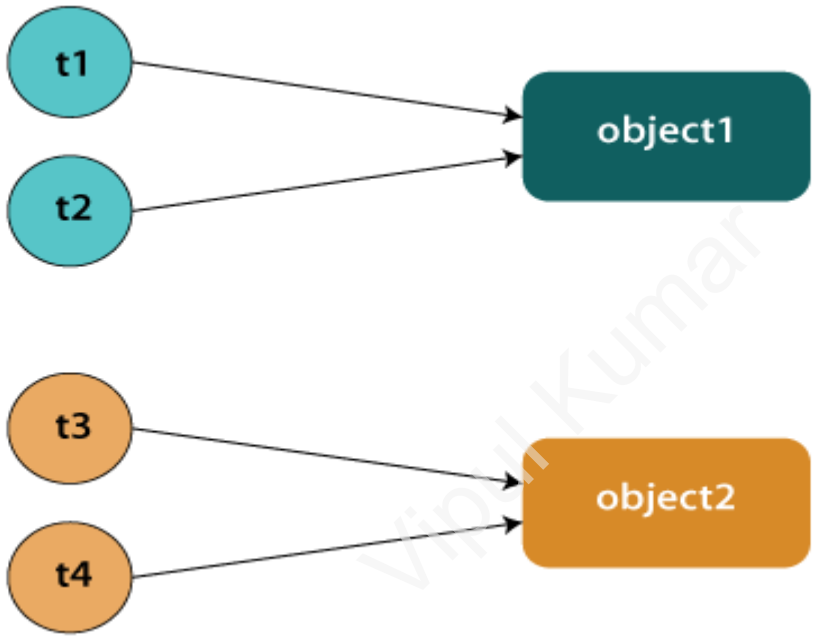
```
synchronized (object reference expression) {
        //code block
}
```

Static Synchronization

If you make any static method as synchronized, the lock will be on the class, not on the object.

Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In the case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refer to a common object that has a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.



A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

```
static void printTable(int n) {
  synchronized (Table.class) {      // Synchronized block on class A
    // ...
  }
}
```

Inter-thread Communication

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of Object class:

- **wait()**
- **notify()**
- **notifyAll()**

1) wait() method

The wait() method causes the current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw an exception.

Method	Description
public final void wait()throws InterruptedException	It waits until the object is notified.

public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor.

```
public final void notify()
```

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

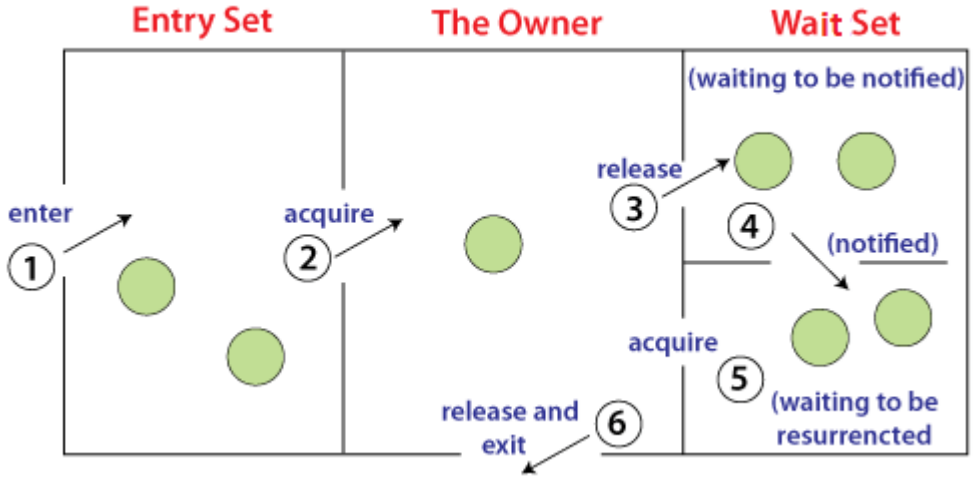
Syntax:

1. `public final void notifyAll()`

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to locks and objects have a lock.

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. **Threads enter to acquire locks.**
2. **Lock is acquired by one thread.**
3. **Now the thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.**
4. **If you call notify() or notifyAll() method, the thread moves to the notified state (runnable state).**
5. **Now thread is available to acquire locks.**
6. **After completion of the task, thread releases the lock and exits the monitor state of the object.**

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()

The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Deadlock in Java

Deadlock can occur in a situation when a thread is waiting for an object lock that is acquired by another thread and the second thread is waiting for an object lock that is acquired by the first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

How to avoid deadlock in code?

A solution for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared resources.

How to Avoid Deadlock in Java?

Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

1. **Avoid Nested Locks: We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.**
2. **Avoid Unnecessary Locks: The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.**
3. **Using Thread Join: A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use join with a maximum time that a thread will take.**

Interrupting a Thread

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the *interrupt()* method on the thread, breaks out the sleeping or waiting state and throws InterruptedException.

If the thread is not in the sleeping or waiting state, calling the *interrupt()* method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.

The 3 methods provided by the Thread class for interrupting a thread

- **public void interrupt() :- discussed above**
- **public static boolean interrupted() :- It returns the interrupted flag after that it sets the flag to false if it is true.**
- **public boolean isInterrupted() :- It returns the interrupted flag either true or false.**

Reentrant Monitor in Java

Java monitors are reentrant means a Java thread can reuse the same monitor for different synchronized methods if a method is called from the method.

Advantage of Reentrant Monitor

It eliminates the possibility of single thread deadlocking.

```
class Reentrant {
   public synchronized void m() {
   n();
   System.out.println("this is m() method");
   }
   public synchronized void n() {
   System.out.println("this is n() method");
   }
}
```


8. Input Output

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform file handling in Java by Java I/O API.

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached to the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

OutputStream

Java applications use an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java applications use an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

OutputStream class

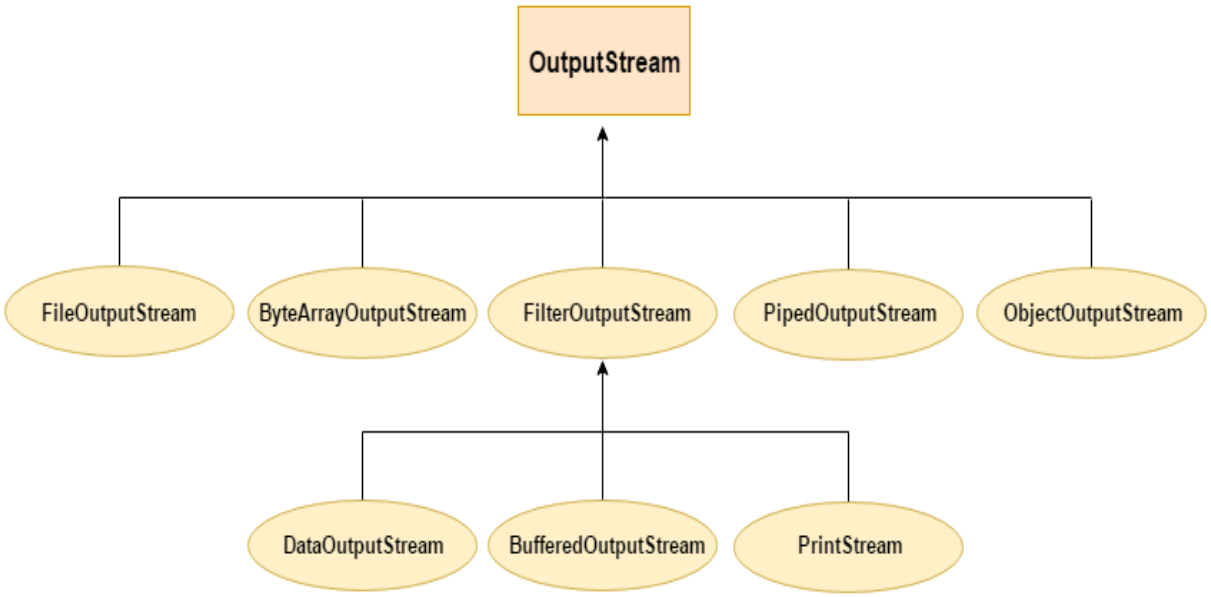
OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

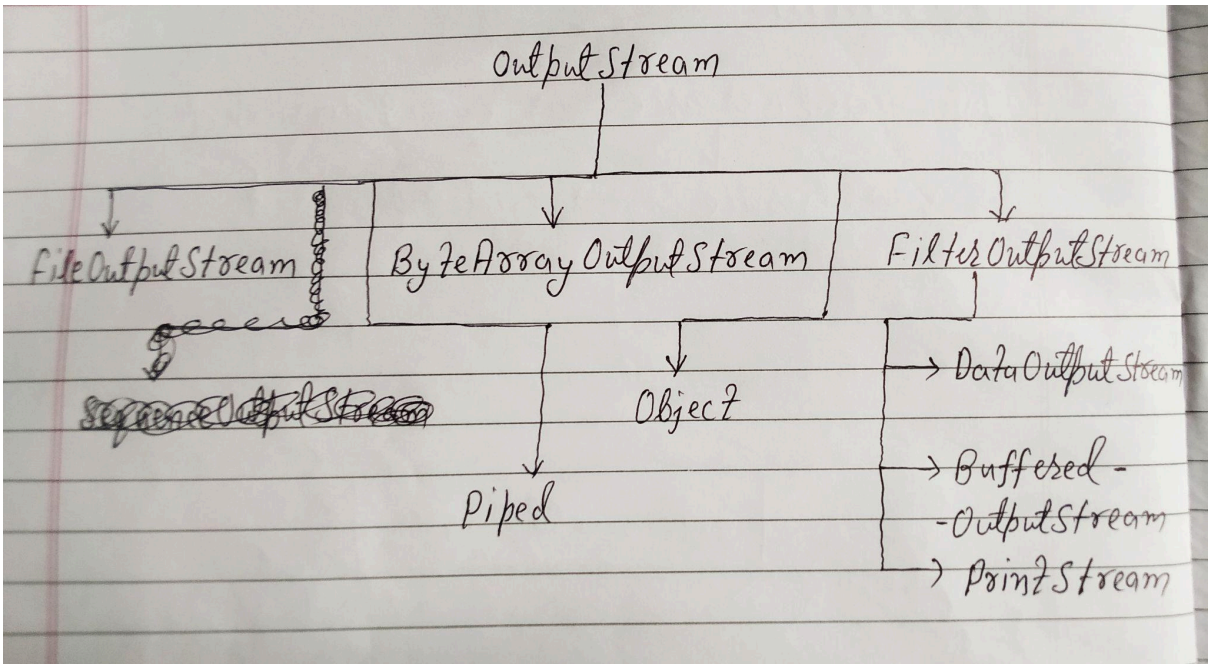
Useful methods of OutputStream

Method	Description

1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy





InputStream class

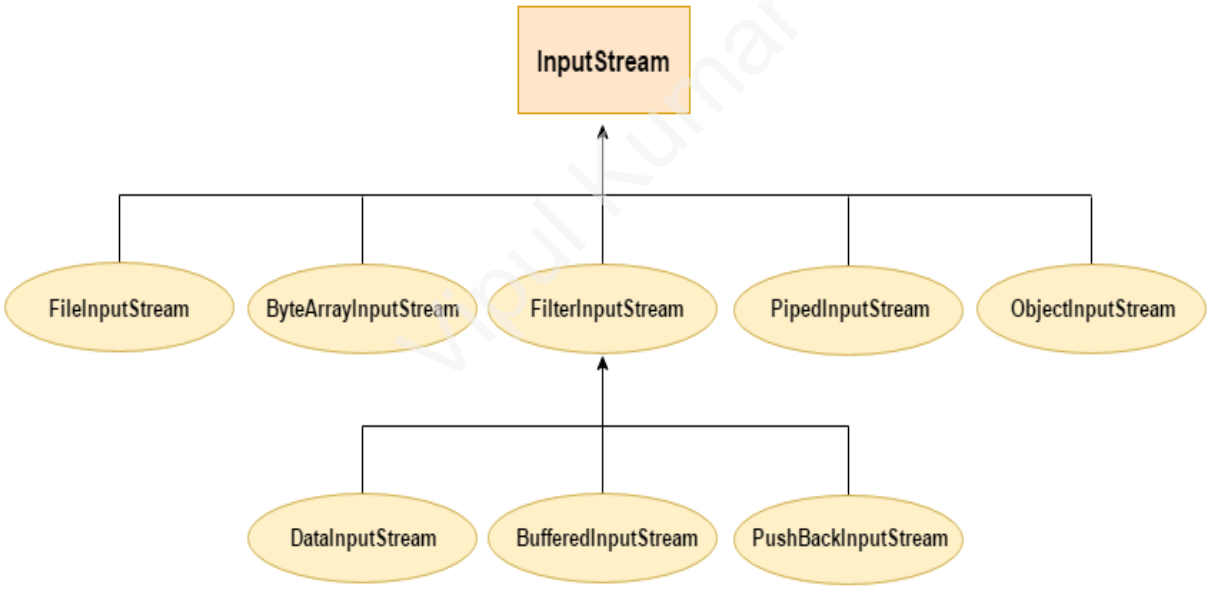
InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

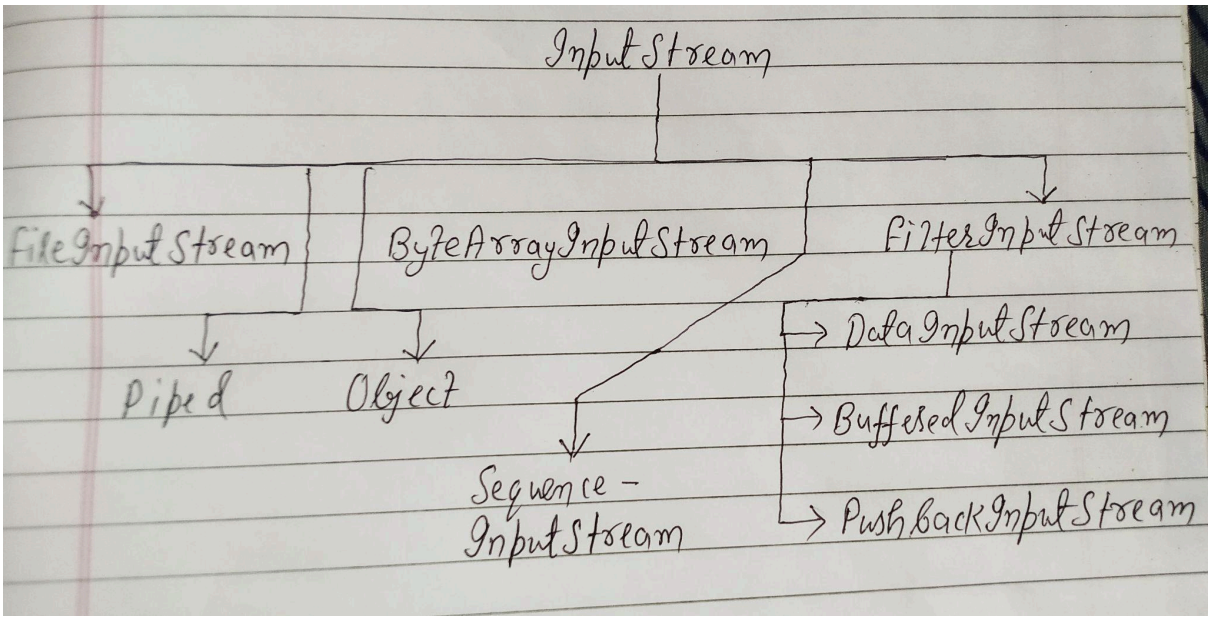
Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.

2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy





FileOutputStream Class

FileOutputStream is an output stream used for writing data to a file.

It is used to write primitive values into a file.

We can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

Syntax:

```
public class FileOutputStream extends OutputStream
```

FileInputStream Class

FileInputStream class obtains input bytes from a file.

It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc.

We can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

```
public class FileInputStream extends InputStream
```

ByteArrayOutputStream Class

Java ByteArrayOutputStream class is used to write common data into multiple files.

```
public class ByteArrayOutputStream extends OutputStream
```

In this stream, the data is written into a byte array which can be written to multiple streams later.

ByteArrayInputStream

The ByteArrayInputStream is composed of two words: ByteArray and InputStream.

The ByteArrayInputStream class contains an internal buffer which is used to read byte arrays as streams. In this stream, the data is read from a byte array.

The buffer of ByteArrayInputStream automatically grows according to data.

```
public class ByteArrayInputStream extends InputStream
```

FilterOutputStream

FilterOutputStream class implements the OutputStream class. It provides different sub classes such as DataOutputStream, BufferedOutputStream and PrintStream to provide additional functionality. So it is less used individually.

```
public class FilterOutputStream extends OutputStream
```

FilterInputStream

The FilterInputStream class implements the InputStream. It contains different sub classes as DataInputStream, BufferedInputStream and PushbackStream for providing additional functionality. So it is less used individually.

```
public class FilterInputStream extends InputStream
```

DataOutputStream

DataOutputStream class allows an application to write primitive Java data types to the output stream in a machine-independent way.

Java applications generally use the data output stream to write data that can later be read by a data input stream.

```
public class DataOutputStream extends FilterOutputStream
implements DataOutput
```


DataInputStream

DataInputStream class allows an application to read primitive data from the input stream in a machine-independent way.

Java applications generally use the data output stream to write data that can later be read by a data input stream.

```
public class DataInputStream extends FilterInputStream implements
DataInput
```

BufferedOutputStream Class

The BufferedOutputStream class is used for buffering an output stream. It internally uses a buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Syntax for adding the buffer in an OutputStream :

```
OutputStream os= new BufferedOutputStream(new FileOutputStream("D:\\IO Package\\testout.txt"));
```

Declaration for Java.io.BufferedOutputStream class:

```
public class BufferedOutputStream extends FilterOutputStream
```

The flush() flushes the data of one stream and sends it into another. It is required if you have connected one stream with another.

BufferedInputStream Class

BufferedInputStream class is used to read information from streams. It internally uses a buffer mechanism to make the performance fast.

The important points about BufferedInputStream are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refills from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.

Declaration for Java.io.BufferedInputStream class:

```
public class BufferedInputStream extends FilterInputStream
```

PrintStream Class

The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

```
public class PrintStream extends FilterOutputStream implements
Closeable. Appendable
```

PushbackInputStream Class

PushbackInputStream class overrides InputStream and provides extra functionality to another input stream. It can unread a byte which is already read and push back one byte.

```
public class PushbackInputStream extends FilterInputStream
```

SequenceInputStream Class

Java SequenceInputStream class is used to read data from multiple streams. It reads data sequentially (one by one).

```
public class SequenceInputStream extends InputStream
```


Writer Class

It is an abstract class for writing to character streams.

The methods that a subclass must implement are write(char[], int, int), flush(), and close(). Most subclasses will override some of the methods defined here to provide higher efficiency, functionality or both.

Reader Class

The Reader is an abstract class for reading character streams. The only methods that a subclass must implement are read(char[], int, int) and close().

Most subclasses, however, will override some of the methods to provide higher efficiency, additional functionality, or both.

Some of the implementation classes are BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader and StringReader.

BufferedWriter Class

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.

```
public class BufferedWriter extends Writer
```

BufferedReader Class

Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by the readLine() method. It makes the performance fast. It inherits Reader class.

```
public class BufferedReader extends Reader
```

CharArrayWriter Class

The CharArrayWriter class can be used to write common data to multiple files. This class inherits the Writer class. Its buffer automatically grows when data is written in this stream. Calling the close() method on this object has no effect.

```
public class CharArrayWriter extends Writer
```

CharArrayReader Class

The CharArrayReader is composed of two words: CharArray and Reader. The CharArrayReader class is used to read character arrays as a reader (stream). It inherits Reader class.

```
public class CharArrayReader extends Reader
```

FilterWriter Class

Java FilterWriter class is an abstract class which is used to write filtered character streams.

The sub class of the FilterWriter should override some of its methods and it may provide additional methods and fields also.

FilterReader Class

Java FilterReader is used to perform filtering operations on reader streams. It is an abstract class for reading filtered character streams.

The FilterReader provides default methods that pass all requests to the contained stream. Subclasses of FilterReader should override some of its methods and may also provide additional methods and fields.

PushbackReader Class

Java PushbackReader class is a character stream reader. It is used to push back a character into stream and overrides the FilterReader class.

```
public class PushbackReader extends FilterReader
```

PipedWriter Class

The PipedWriter class is used to write java pipes as a stream of characters. This class is used generally for writing text. Generally PipedWriter is connected to a PipedReader and used by different threads.

PipedReader Class

The PipedReader class is used to read the contents of a pipe as a stream of characters. This class is used generally to read text.

The PipedReader class must be connected to the same Piped and are used by different threads.

StringWriter Class

Java StringWriter class is a character stream that collects output from a string buffer, which can be used to construct a string. The StringWriter class inherits the Writer class.

In the StringWriter class, system resources like network socket and files are not used, therefore closing the StringWriter is not necessary.


```
public class StringWriter extends Writer
```

StringReader Class

Java StringReader class is a character stream with string as a source. It takes an input string and changes it into a character stream. It inherits Reader class.

In the StringReader class, system resources like network sockets and files are not used, therefore closing the StringReader is not necessary.

```
public class StringReader extends Reader
```

PrintWriter Class

Java PrintWriter class is the implementation of Writer class. It is used to print the formatted representation of objects to the text-output stream.

```
public class PrintWriter extends Writer
```

OutputStreamWriter Class

OutputStreamWriter is a class which is used to convert character stream to byte stream, the characters are encoded into byte using a specified charset. write() method calls the encoding converter which converts the character into bytes. The resulting bytes are then accumulated in a buffer before being written into the underlying output stream. The characters passed to write() methods are not buffered. We optimise the performance of OutputStreamWriter by using it within a BufferedWriter so that to avoid frequent converter invocation.

InputStreamReader Class

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

FileWriter Class

Java FileWriter class is used to write character-oriented data to a file. It is a character-oriented class which is used for file handling in java.

Unlike FileOutputStream class, you don't need to convert string into byte array because it provides a method to write string directly.

```
public class FileWriter extends OutputStreamWriter
```

FileReader Class

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

It is a character-oriented class which is used for file handling in java.

```
public class FileReader extends InputStreamReader
```

Console Class

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

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

```
public final class Console extends Object implements Flushable
```

System class provides a static method console() that returns the singleton instance of Console class.

```
public static Console console(){}
```

Let's see the code to get the instance of Console class.

```
Console c=System.console();
```

FilePermission Class

Java FilePermission class contains the permission related to a directory or file. All the permissions are related to the path. The path can be of two types:

1) D:\\IO\\-: It indicates that the permission is associated with all sub directories and files recursively.

2) D:\\IO*: It indicates that the permission is associated with all directory and files within this directory excluding subdirectories.

```
public final class FilePermission extends Permission implements
Serializable
```

File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class has several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

FileDescriptor Class

FileDescriptor class serves as a handle to the underlying machine-specific structure representing an open file, an open socket, or another source or sink of bytes. The handle can be err, in or out.

The FileDescriptor class is used to create a FileInputStream or FileOutputStream to contain it.

Modifier	Type	Field	Description
static	FileDescriptor	err	A handle to the standard error stream.
static	FileDescriptor	in	A handle to the standard input stream.

static	FileDescriptor	out	A handle to the standard output stream.

Constructor	Description
FileDescriptor()	Constructs an (invalid) FileDescriptor object.

Modifier and Type	Method	Description
void	sync()	It forces all system buffers to synchronize with the underlying device.
boolean	valid()	It tests if this file descriptor object is valid.

RandomAccessFile Class

This class is used for reading and writing to random access files. A random access file behaves like a large array of bytes. There is a cursor implied to the array called file pointer, by moving the cursor we do the read write

operations. If end-of-file is reached before the desired number of bytes has been read then EOFException is thrown. It is a type of IOException.

Scanner Class

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

ObjectStream Class

ObjectStreamClass acts as a Serialization descriptor for class. This class contains the name and serialVersionUID of the class.

ObjectStreamField Class

A description of a Serializable field from a Serializable class. An array of ObjectStreamFields is used to declare the Serializable fields of a class.

The java.io.ObjectStreamClass.getField(String name) method gets the field of this class by name.

public char getTypeCode()

Returns character encoding of field type.

Serialization and Deserialization

Serialization in Java is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called *deserialization* where a byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform.

For serializing the object, we call the writeObject() method of the ObjectOutputStream class, and for deserialization we call the readObject() method of *ObjectInputStream* class.

We must have to implement the *Serializable* interface for serializing the object.

It is mainly used to travel object's state on the network (that is known as marshalling).

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The Cloneable and Remote are also marker interfaces.

The Serializable interface must be implemented by the class whose object needs to be persisted.

The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

If a class implements Serializable interface then all its sub classes will also be serializable.

The SerializeISA class has serialized the Student class object that extends the Person class which is Serializable. Parent class properties are inherited to subclasses so if the parent class is Serializable, subclasses would also be.

If a class has a reference to another class, all the references must be Serializable otherwise the serialization process will not be performed. In such cases, *NotSerializableException* is thrown at runtime.

Note: All the objects within an object must be Serializable.

If there is any static data member in a class, it will not be serialized because static is the part of the class not the object.

Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serialiizable, serialization will fail.

The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.

The Externalizable interface provides two methods:

- **public void writeExternal(ObjectOutput out) throws IOException**
- **public void readExternal(ObjectInput in) throws IOException**

If you don't want to serialize any data member of a class, you can mark it as transient.

Now, id will not be serialized, so when you deserialize the object after serialization, you will not get the value of id. It will always return the default value. In such a case, it will return 0 because the data type of id is an integer.

SerialVersionUID

The serialization process at runtime associates an id with each Serializable class which is known as SerialVersionUID. It is used to verify the sender and receiver of the serialized object. The sender and receiver must be the same. To verify it, SerialVersionUID is used. The sender and receiver must have the same SerialVersionUID, otherwise, InvalidClassException will be thrown when you deserialize the object. We can also declare our own SerialVersionUID in the Serializable class. To do so, you need to create a field SerialVersionUID and assign a value to it. It must be of the long type with static and final. It is suggested to explicitly declare the serialVersionUID field in the class and have it private also. For example:

```
private static final long serialVersionUID=1L;
```

Java transient Keyword

In Java, Serialization is used to convert an object into a stream of the byte. The byte stream consists of the data of the instance as well as the type of data stored in that instance. Deserialization performs exactly the opposite operation. It converts the byte sequence into original object data. During the serialization, when we do not want an object to be serialized we can use a transient keyword.

Why use the transient keyword?

The transient keyword can be used with the data members of a class in order to avoid their serialization. For example, if a program accepts a user's login details and password. But we don't want to store the original password in the file. Here, we can use the transient keyword and when JVM reads the transient keyword it ignores the original value of the object and instead stores the default value of the object.

Syntax:

```
private transient <member variable>;
```

Or

```
transient private <member variable>;
```

When to use the transient keyword?

1. **The transient modifier can be used where there are data members derived from the other data members within the same instance of the class.**
2. **This transient keyword can be used with the data members which do not depict the state of the object.**

3. **The data members of a non-serialized object or class can use a transient modifier.**

Example of Java Transient Keyword

Let's take an example, we have declared a class as *Student*, it has three data members *id, name* and *age*. If you serialize the object, all the values will be serialized but we don't want to serialize one value, e.g. *age* then we can declare the age data member as transient.

In this example, we have created two classes: Student and *PersistExample*. The *age* data member of the *Student* class is declared as transient, its value will not be serialized.

If you deserialize the object, you will get the default value for the transient variable.

9. Conversion

Convert String to int

We can convert String to an int in java using Integer.parseInt() method. To convert a String into an Integer, we can use the Integer.valueOf() method which returns an instance of Integer class.

```
public static int parseInt(String s)

int i=Integer.parseInt("200");

Integer i=Integer.valueOf("200");
```

If you don't have numbers in string literal, calling Integer.parseInt() or Integer.valueOf() methods throw a NumberFormatException.

Convert int to String

We can convert int to String in java using *String.valueOf()* and *Integer.toString()* methods. Alternatively, we can use the String.format() method, string concatenation operator etc.

```
public static String valueOf(int i)

public static String toString(int i)

public static String format(String format, Object... args)
```

Convert String to long

We can convert String to long in java using *Long.parseLong()* method.

```
public static long parseLong(String s)
```


Convert long to String

We can convert long to String in java using *String.valueOf()* and *Long.toString()* methods.

```
public static String valueOf(long i)

public static String toString(long i)
```

Convert String to Float

We can convert String to float in java using *Float.parseFloat()* method.

```
public static int parseFloat(String s)
```

Convert Float to String

We can convert float to String in java using *String.valueOf()* and *Float.toString()* methods.

```
public static String valueOf(float f)

public static String toString(float f)
```

Convert String to double

We can convert String to double in java using *Double.parseDouble()* method.

```
public static double parseDouble(String s)
```

Convert Double to String

We can convert double to String in java using *String.valueOf()* and *Double.toString()* methods.

```
public static String valueOf(double d)

public static String toString(double d)
```


Convert String to Date

We can convert String to Date in java using the *parse()* method of DateFormat and SimpleDateFormat classes.

Convert Date to String

We can convert Date to String in java using *format()* method of java.text.DateFormat class.

```
String format(Date d)
```

Convert String to char

We can convert String to char in java using *charAt()* method of String class.

```
public char charAt(int index)
```

String to char in java using toCharArray() method. The toCharArray() method of String class converts this string into a character array.

Convert char to String

We can convert char to String in java using *String.valueOf(char)* method of String class and *Character.toString(char)* method of Character class.

We can convert char array to String in java using *String.valueOf(char[])* method of String class

Convert String to Object

We can convert String to Object in java with assignment operator. Each class is internally a child class of the Object class. So you can assign strings to objects directly.

You can also convert String to Class type object using Class.forName() method.

Convert Object to String

We can convert an Object to a String in Java using the toString() method of Object class or String.valueOf(object) method.

Convert int to long

We can convert int to long in java using assignment operator. There is nothing to do extra because lower type can be converted to higher type implicitly.

It is also known as implicit type casting or type promotion.

We can convert int value to Long object by instantiating Long class or calling Long.valueOf() method.

Convert long to int

We can convert long to int in java using type casting. To convert higher data type into lower, we need to perform typecasting.

Type Casting in java is performed through typecast operator *(datatype)*.

We can convert a Long object to int by the intValue() method of Long class.
Let's see the simple code to convert Long to int in java.

Convert int to double

We can convert int to double in java using assignment operator. There is nothing to do extra because lower type can be converted to higher type implicitly.

It is also known as implicit type casting or type promotion.

We can convert int value to Double object by instantiating Double class or calling Double.valueOf() method.

```
Double d= new Double(i);//first way

Double d2=Double.valueOf(i);//second way
```

Convert double to int

We can convert double to int in java using type casting. To convert double data type into int, we need to perform typecasting.

Type Casting in java is performed through typecast operator *(datatype)*.

We can convert a Double object to int by intValue() method of Double class.

Convert char to int

We can convert char to int in java using various ways. If we directly assign a char variable to int, it will return the ASCII value of the given character.

If a char variable contains an int value, we can get the int value by calling the Character.getNumericValue*(char)* method. Alternatively, we can use the String.valueOf(char) method.

Convert int to char

We can convert int to char in java using type casting. To convert higher data type into lower, we need to perform typecasting. Here, the ASCII character of integer value will be stored in the char variable.

To get the actual value in char variable, you can add '0' with int variable. Alternatively, you can use the Character.forDigit() method.

Convert String to boolean

We can convert String to boolean in java using *Boolean.parseBoolean(string)* method.

To convert a String into a Boolean object, we can use the Boolean.valueOf*(string)* method which returns an instance of Boolean class.

To get boolean true, the string must contain "true". Here, the case is ignored. So, "true" or "TRUE" will return boolean true. Any other string value except "true" returns boolean false.

Convert boolean to String

We can convert boolean to String in java using *String.valueOf(boolean)* method.

Alternatively, we can use *Boolean.toString(boolean)* method which also converts boolean into String.

Convert Date to Timestamp

We can convert Date to Timestamp in java using the constructor *of java.sql.Timestamp class*.

The constructor of the Timestamp class receives long value as an argument. So you need to convert the date into a long value using the getTime() *method of the java.util.Date class*.

We can also format the output of Timestamp using java.text.SimpleDateFormat class.

```
Date date = new Date();
Timestamp ts=new Timestamp(date.getTime());
SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
System.out.println(formatter.format(ts));
```


Convert Timestamp to Date

We can convert Timestamp to Date in java using the constructor *of java.util.Date class*.The constructor of Date class receives a long value as an argument. So, you need to convert a Timestamp object into a long value using the getTime() *method of java.sql.Timestamp class*.

```
Timestamp ts=new Timestamp(System.currentTimeMillis());
Date date=new Date(ts.getTime());
```

The Timestamp class extends Date class. So, you can directly assign instances of Timestamp class into Date. In such a case, the output of the Date object will be like a Timestamp. But, it is not suggested by Java Doc because you may lose the milliseconds or nanoseconds of data.

```
Date date=ts;
```

Convert Binary to Decimal

We can convert binary to decimal in java using *Integer.parseInt()* method or custom logic.

```
public static int parseInt(String s,int radix)
```

Convert Decimal to Binary

We can convert decimal to binary in java using *Integer.toBinaryString()* method or custom logic.

```
public static String toBinaryString(int decimal)
```


Convert Hexadecimal to Decimal

We can convert hexadecimal to decimal in java using *Integer.parseInt()* method or custom logic.

```
public static int parseInt(String s,int radix)
```

Convert Decimal to Hexadecimal

We can convert decimal to hexadecimal in java using *Integer.toHexString()* method or custom logic.

```
public static String toHexString(int decimal)
```

Convert Octal to Decimal

We can convert octal to decimal in java using *Integer.parseInt()* method or custom logic.

```
public static int parseInt(String s,int radix)
```

Convert Decimal to Octal

We can convert decimal to octal in java using *Integer.toOctalString()* method or custom logic.

```
public static String toOctalString(int decimal)
```

10. Collection

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects. Java Collections can achieve all the operations that you perform on data such as searching, sorting, insertion, manipulation, and deletion.

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

What is Collection framework

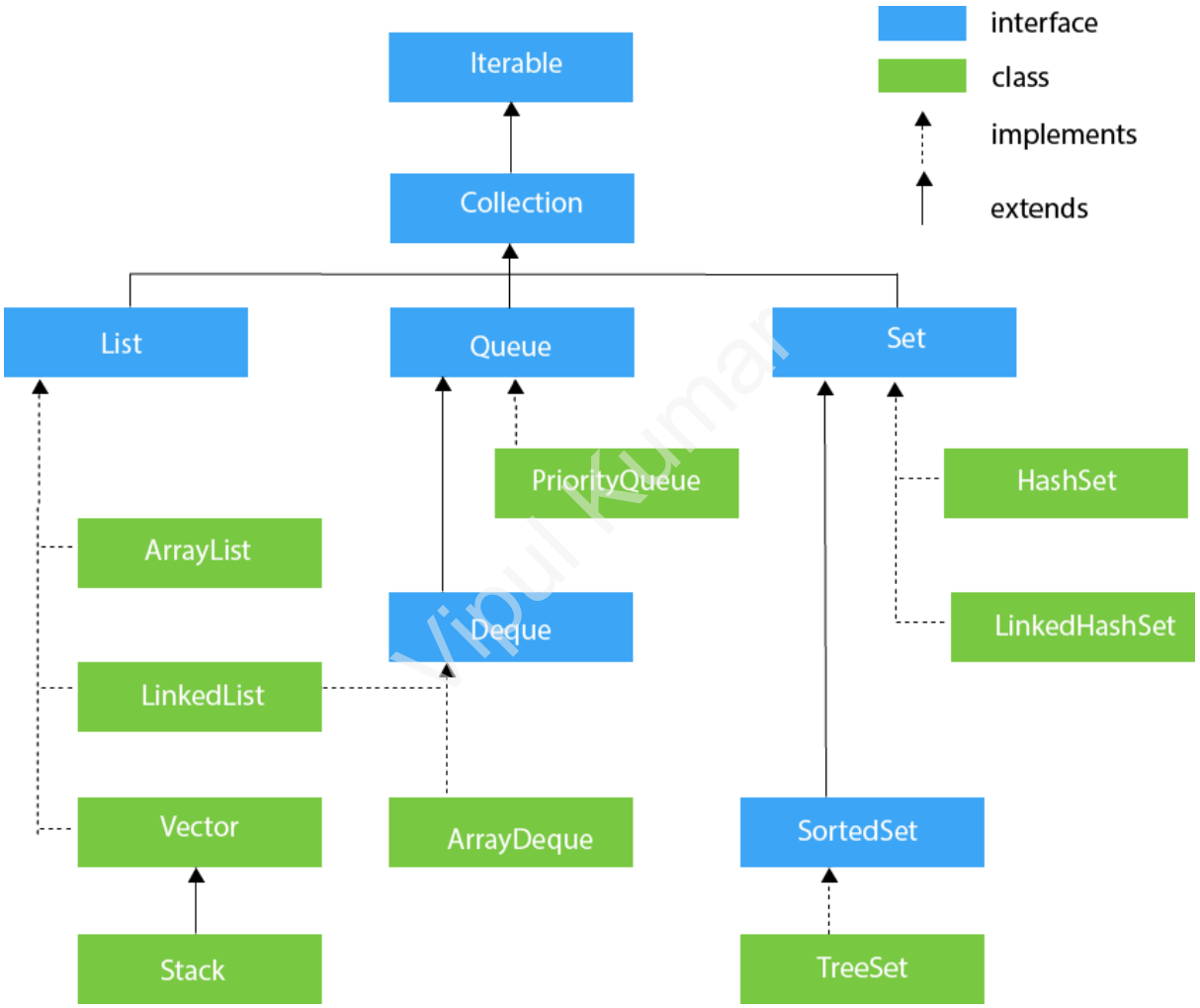
The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes

2. **Algorithm**

Hierarchy of Collection Framework

The java.util package contains all the classes and interfaces for the Collection framework.



Collection Interface

- **The Collection interface is the interface which is implemented by all the classes in the collection framework.**
- **The Collection interface builds the foundation on which the collection framework depends.**

There are various ways to traverse the collection elements:

1. **By Iterator interface.**
2. **By for-each loop.**
3. **By ListIterator interface.**
4. **By for-loop.**
5. **By forEach() method.**
6. **By forEachRemaining() method.**

```
PS C:\Users\hp> javap java.util.Collection
Compiled from "Collection.java"
public interface java.util.Collection<E> extends java.lang.Iterable<E> {
  public abstract int size();
  public abstract boolean isEmpty();
  public abstract boolean contains(java.lang.Object);
  public abstract java.util.Iterator<E> iterator();
  public abstract java.lang.Object[] toArray();
  public abstract <T> T[] toArray(T[]);
  public default <T> T[] toArray(java.util.function.IntFunction<T[]>);
  public abstract boolean add(E);
  public abstract boolean remove(java.lang.Object);
  public abstract boolean containsAll(java.util.Collection<?>);
  public abstract boolean addAll(java.util.Collection<? extends E>);
  public abstract boolean removeAll(java.util.Collection<?>);
  public default boolean removeIf(java.util.function.Predicate<? super E>);
  public abstract boolean retainAll(java.util.Collection<?>);
  public abstract void clear();
  public abstract boolean equals(java.lang.Object);
  public abstract int hashCode();
  public default java.util.Spliterator<E> spliterator();
  public default java.util.stream.Stream<E> stream();
  public default java.util.stream.Stream<E> parallelStream();
}
PS C:\Users\hp> _
```


Iterator interface

- **The Iterator interface provides the facility of iterating the elements in a forward direction only.**
- **The Java Iterator is also known as the universal cursor of Java as it is appropriate for all the classes of the Collection framework.**
- **In Java Iterator, we can use both of the read and remove operations.**

- If a user is working with a for loop, they cannot modernize(add/remove) the Collection, whereas, if they use the Java Iterator, they can simply update the Collection.
- The replacement and extension of a new component are not approved by the Java Iterator.
- In CRUD Operations, the Java Iterator does not hold the various operations like CREATE and UPDATE.
- In comparison with the Spliterator, Java Iterator does not support traversing elements in the parallel pattern which implies that Java Iterator supports only Sequential iteration.
- In comparison with the Spliterator, Java Iterator does not support more reliable execution to traverse the bulk volume of data.


```
PS C:\Users\hp> javap java.util.Iterator
Compiled from "Iterator.java"
public interface java.util.Iterator<E> {
  public abstract boolean hasNext();
  public abstract E next();
  public default void remove();
  public default void forEachRemaining(java.util.function.Consumer<? super E>);
}
PS C:\Users\hp>
```

Iterable Interface

- The Iterable interface is the root interface for all the collection classes.
- The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

```
PS C:\Users\hp> javap java.lang.Iterable
Compiled from "Iterable.java"
public interface java.lang.Iterable<T> {
  public abstract java.util.Iterator<T> iterator();
  public default void forEach(java.util.function.Consumer<? super T>);
  public default java.util.Spliterator<T> spliterator();
}
PS C:\Users\hp>
```

List Interface

- List interface is the child interface of Collection interface.

- It inhibits a list type data structure in which we can store the ordered collection of objects.
- It can have duplicate values. We can convert the List to Array by calling the list.toArray() method.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= new ArrayList();
2. List <data-type> list2 = new LinkedList();
3. List <data-type> list3 = new Vector();
4. List <data-type> list4 = new Stack();

```
PS C:\Users\hp> javap java.util.List
Compiled from "List.java"
public interface java.util.List<E> extends java.util.Collection<E> {
  public abstract int size();
  public abstract boolean isEmpty();
  public abstract boolean contains(java.lang.Object);
  public abstract java.util.Iterator<E> iterator();
  public abstract java.lang.Object[] toArray();
  public abstract <T> T[] toArray(T[]);
  public abstract boolean add(E);
  public abstract boolean remove(java.lang.Object);
  public abstract boolean containsAll(java.util.Collection<?>);
  public abstract boolean addAll(java.util.Collection<? extends E>);
  public abstract boolean addAll(int, java.util.Collection<? extends E>);
  public abstract boolean removeAll(java.util.Collection<?>);
  public abstract boolean retainAll(java.util.Collection<?>);
  public default void replaceAll(java.util.function.UnaryOperator<E>);
  public default void sort(java.util.Comparator<? super E>);
  public abstract void clear();
  public abstract boolean equals(java.lang.Object);
  public abstract int hashCode();
  public abstract E get(int);
  public abstract E set(int, E);
  public abstract void add(int, E);
  public abstract E remove(int);
  public abstract int indexOf(java.lang.Object);
  public abstract int lastIndexOf(java.lang.Object);
  public abstract java.util.ListIterator<E> listIterator();
  public abstract java.util.ListIterator<E> listIterator(int);
  public abstract java.util.List<E> subList(int, int);
  public default java.util.Spliterator<E> spliterator();
  public static <E> java.util.List<E> of();
  public static <E> java.util.List<E> of(E);
  public static <E> java.util.List<E> of(E, E);
  public static <E> java.util.List<E> of(E, E, E);
  public static <E> java.util.List<E> of(E, E, E, E);
  public static <E> java.util.List<E> of(E, E, E, E, E);
  public static <E> java.util.List<E> of(E, E, E, E, E, E);
  public static <E> java.util.List<E> of(E, E, E, E, E, E, E);
  public static <E> java.util.List<E> of(E, E, E, E, E, E, E, E);
  public static <E> java.util.List<E> of(E, E, E, E, E, E, E, E, E);
  public static <E> java.util.List<E> of(E, E, E, E, E, E, E, E, E, E);
  public static <E> java.util.List<E> of(E...);
  public static <E> java.util.List<E> copyOf(java.util.Collection<? extends E>);
}
PS C:\Users\hp>
```


Java ListIterator Interface

ListIterator Interface is used to traverse the element in a backward and forward direction.

```
PS C:\Users\hp> javap java.util.ListIterator
Compiled from "ListIterator.java"
public interface java.util.ListIterator<E> extends java.util.Iterator<E> {
  public abstract boolean hasNext();
  public abstract E next();
  public abstract boolean hasPrevious();
  public abstract E previous();
  public abstract int nextIndex();
  public abstract int previousIndex();
  public abstract void remove();
  public abstract void set(E);
  public abstract void add(E);
}
PS C:\Users\hp> _
```

ArrayList

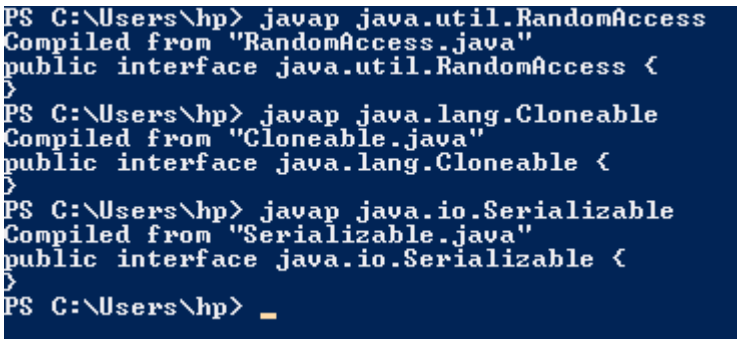
- **The ArrayList class implements the List interface.**
- **It uses a dynamic array to store the duplicate element of different data types.**
- **The ArrayList class maintains the insertion order and is non-synchronized.**
- **The elements stored in the ArrayList class can be randomly accessed.**
- **It inherits the AbstractList class and implements List Interface.**
- **Java ArrayList class can contain duplicate elements.**
- **In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.**
- **The *java.util* package provides a utility class Collections, which has the static method sort(). Using the Collections.sort() method, we can easily sort the ArrayList.**
- **Capacity represents the total number of elements the array list can contain. Size represents the total number of elements present in the array. Therefore, the capacity of an array list is always greater than or equal to the size of the array list. When we add an element to the array list, it checks whether the size of the array list has become equal to the capacity or not. If yes, then the capacity of the array list increases. The default capacity of the array list is 10.**

```
PS C:\Users\hp> javap java.util.ArrayList
Compiled from "ArrayList.java"
public class java.util.ArrayList<E> extends java.util.AbstractList<E> implements java.util.List<E>, java.util.RandomAccess,
 java.lang.Cloneable, java.io.Serializable {
  transient java.lang.Object[] elementData;
  public java.util.ArrayList(int);
  public java.util.ArrayList();
  public java.util.ArrayList(java.util.Collection<? extends E>);
  public void trimToSize();
  public void ensureCapacity(int);
  public int size();
  public boolean isEmpty();
  public boolean contains(java.lang.Object);
  public int indexOf(java.lang.Object);
  int indexOfRange(java.lang.Object, int, int);
  public int lastIndexOf(java.lang.Object);
  int lastIndexOfRange(java.lang.Object, int, int);
  public java.lang.Object clone();
  public java.lang.Object[] toArray();
  public <T> T[] toArray(T[]);
  E elementData(int);
  static <E> E elementAt(java.lang.Object[], int);
  public E get(int);
  public E set(int, E);
  public boolean add(E);
  public void add(int, E);
  public E remove(int);
  public boolean equals(java.lang.Object);
  boolean equalsRange(java.util.List<?>, int, int);
  public int hashCode();
  int hashCodeRange(int, int);
  public boolean remove(java.lang.Object);
  public void clear();
  public boolean addAll(java.util.Collection<? extends E>);
  public boolean addAll(int, java.util.Collection<? extends E>);
  protected void removeRange(int, int);
  public boolean removeAll(java.util.Collection<?>);
  public boolean retainAll(java.util.Collection<?>);
  boolean batchRemove(java.util.Collection<?>, boolean, int, int);
  public java.util.ListIterator<E> listIterator(int);
  public java.util.ListIterator<E> listIterator();
  public java.util.Iterator<E> iterator();
  public java.util.List<E> subList(int, int);
  public void forEach(java.util.function.Consumer<? super E>);
  public java.util.Spliterator<E> spliterator();
  public boolean removeIf(java.util.function.Predicate<? super E>);
  boolean removeIf(java.util.function.Predicate<? super E>, int, int);
  public void replaceAll(java.util.function.UnaryOperator<E>);
  public void sort(java.util.Comparator<? super E>);
  void checkInvariants();
  static {};
}
PS C:\Users\hp>
```

AbstractList & AbstractCollection

```
PS C:\Users\hp> javap java.util.AbstractList
Compiled from "AbstractList.java"
public abstract class java.util.AbstractList<E> extends java.util.AbstractCollection<E> implements java.util.List<E> {
  protected transient int modCount;
  protected java.util.AbstractList();
  public boolean add(E);
  public abstract E get(int);
  public E set(int, E);
  public void add(int, E);
  public E remove(int);
  public int indexOf(java.lang.Object);
  public int lastIndexOf(java.lang.Object);
  public void clear();
  public boolean addAll(int, java.util.Collection<? extends E>);
  public java.util.Iterator<E> iterator();
  public java.util.ListIterator<E> listIterator();
  public java.util.ListIterator<E> listIterator(int);
  public java.util.List<E> subList(int, int);
  static void subListRangeCheck(int, int, int);
  public boolean equals(java.lang.Object);
  public int hashCode();
  protected void removeRange(int, int);
}
PS C:\Users\hp> javap java.util.AbstractCollection
Compiled from "AbstractCollection.java"
public abstract class java.util.AbstractCollection<E> implements java.util.Collection<E> {
  protected java.util.AbstractCollection();
  public abstract java.util.Iterator<E> iterator();
  public abstract int size();
  public boolean isEmpty();
  public boolean contains(java.lang.Object);
  public java.lang.Object[] toArray();
  public <T> T[] toArray(T[]);
  public boolean add(E);
  public boolean remove(java.lang.Object);
  public boolean containsAll(java.util.Collection<?>);
  public boolean addAll(java.util.Collection<? extends E>);
  public boolean removeAll(java.util.Collection<?>);
  public boolean retainAll(java.util.Collection<?>);
  public void clear();
  public java.lang.String toString();
}
PS C:\Users\hp> _
```

Marker Interface(RandomAccess, Serializable, Cloneable)



LinkedList

- LinkedList implements the Collection interface.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements. It maintains the insertion order and is not synchronized.
- In LinkedList, the manipulation is fast because no shifting is required.
- It inherits the AbstractSequentialList class and implements List and Deque interfaces.
- Java LinkedList class can be used as a list, stack or queue.

```
public class java.util.LinkedList<E> extends java.util.AbstractSequentialList<E> implements java.util.List<E>,
java.util.Deque<E>, java.lang.Cloneable, java.io.Serializable {
  transient int size;
  transient java.util.LinkedList$Node<E> first;
  transient java.util.LinkedList$Node<E> last;
  public java.util.LinkedList();
  public java.util.LinkedList(java.util.Collection<? extends E>);
  void linkLast(E);
  void linkBefore(E, java.util.LinkedList$Node<E>);
  E unlink(java.util.LinkedList$Node<E>);
  public E getFirst();
  public E getLast();
  public E removeFirst();
  public E removeLast();
  public void addFirst(E);
  public void addLast(E);
  public boolean contains(java.lang.Object);
  public int size();
  public boolean add(E);
  public boolean remove(java.lang.Object);
  public boolean addAll(java.util.Collection<? extends E>);
  public boolean addAll(int, java.util.Collection<? extends E>);
  public void clear();
  public E get(int);
  public E set(int, E);
  public void add(int, E);
  public E remove(int);
  java.util.LinkedList$Node<E> node(int);
  public int indexOf(java.lang.Object);
  public int lastIndexOf(java.lang.Object);
  public E peek();
  public E element();
  public E poll();
  public E remove();
  public boolean offer(E);
  public boolean offerFirst(E);
  public boolean offerLast(E);
  public E peekFirst();
  public E peekLast();
  public E pollFirst();
  public E pollLast();
  public void push(E);
  public E pop();
  public boolean removeFirstOccurrence(java.lang.Object);
  public boolean removeLastOccurrence(java.lang.Object);
  public java.util.ListIterator<E> listIterator(int);
  public java.util.Iterator<E> descendingIterator();
  public java.lang.Object clone();
  public java.lang.Object[] toArray();
  public <T> T[] toArray(T[]);
  public java.util.Spliterator<E> spliterator();
}
PS C:\Users\hp> _
```


AbstractSequentialList

```
PS C:\Users\hp> javap java.util.AbstractSequentialList
Compiled from "AbstractSequentialList.java"
public abstract class java.util.AbstractSequentialList<E> extends java.util.AbstractList<E> {
  protected java.util.AbstractSequentialList();
  public E get(int);
  public E set(int, E);
  public void add(int, E);
  public E remove(int);
  public boolean addAll(int, java.util.Collection<? extends E>);
  public java.util.Iterator<E> iterator();
  public abstract java.util.ListIterator<E> listIterator(int);
}
PS C:\Users\hp> _
```

Difference Between ArrayList and LinkedList

ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.

However, there are many differences between the ArrayList and LinkedList classes that are given below.

ArrayList	LinkedList

1) ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the list interface.

Vector

- **Vector uses a dynamic array to store the data elements. It is similar to ArrayList.**
- **However, It is synchronized and contains many methods that are not part of the Collection framework.**
- **The Iterators returned by the Vector class are *fail-fast*. In case of concurrent modification, it fails and throws the ConcurrentModificationException.**
- **It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such cases.**

```
PS C:\Users\hp> javap java.util.Vector
Compiled from "Vector.java"
public class java.util.Vector<E> extends java.util.AbstractList<E> implements java.util.List<E>, java.util.Rand
omAccess, java.lang.Cloneable, java.io.Serializable {
  protected java.lang.Object[] elementData;
  protected int elementCount;
  protected int capacityIncrement;
  public java.util.Vector(int, int);
  public java.util.Vector(int);
  public java.util.Vector();
  public java.util.Vector(java.util.Collection<? extends E>);
  public synchronized void copyInto(java.lang.Object[]);
  public synchronized void trimToSize();
  public synchronized void ensureCapacity(int);
  public synchronized void setSize(int);
  public synchronized int capacity();
  public synchronized int size();
  public synchronized boolean isEmpty();
  public java.util.Enumeration<E> elements();
  public boolean contains(java.lang.Object);
  public int indexOf(java.lang.Object);
  public synchronized int indexOf(java.lang.Object, int);
  public synchronized int lastIndexOf(java.lang.Object);
  public synchronized int lastIndexOf(java.lang.Object, int);
  public synchronized E elementAt(int);
  public synchronized E firstElement();
  public synchronized E lastElement();
  public synchronized void setElementAt(E, int);
  public synchronized void removeElementAt(int);
  public synchronized void insertElementAt(E, int);
  public synchronized void addElement(E);
  public synchronized boolean removeElement(java.lang.Object);
  public synchronized void removeAllElements();
  public synchronized java.lang.Object clone();
  public synchronized java.lang.Object[] toArray();
  public synchronized <T> T[] toArray(T[]);
  E elementData(int);
  static <E> E elementAt(java.lang.Object[], int);
  public synchronized E get(int);
  public synchronized E set(int, E);
  public synchronized boolean add(E);
  public boolean remove(java.lang.Object);
  public void add(int, E);
  public synchronized E remove(int);
  public void clear();
  public synchronized boolean containsAll(java.util.Collection<?>);
  public boolean addAll(java.util.Collection<? extends E>);
  public boolean removeAll(java.util.Collection<?>);
  public boolean retainAll(java.util.Collection<?>);
  public boolean removeIf(java.util.function.Predicate<? super E>);
  public synchronized boolean addAll(int, java.util.Collection<? extends E>);
  public synchronized boolean equals(java.lang.Object);
  public synchronized int hashCode();
  public synchronized java.lang.String toString();
  public synchronized java.util.List<E> subList(int, int);
```

```
  public synchronized boolean add(E);
  public boolean remove(java.lang.Object);
  public void add(int, E);
  public synchronized E remove(int);
  public void clear();
  public synchronized boolean containsAll(java.util.Collection<?>);
  public boolean addAll(java.util.Collection<? extends E>);
  public boolean removeAll(java.util.Collection<?>);
  public boolean retainAll(java.util.Collection<?>);
  public boolean removeIf(java.util.function.Predicate<? super E>);
  public synchronized boolean addAll(int, java.util.Collection<? extends E>);
  public synchronized boolean equals(java.lang.Object);
  public synchronized int hashCode();
  public synchronized java.lang.String toString();
  public synchronized java.util.List<E> subList(int, int);
  protected synchronized void removeRange(int, int);
  public synchronized java.util.ListIterator<E> listIterator(int);
  public synchronized java.util.ListIterator<E> listIterator();
  public synchronized java.util.Iterator<E> iterator();
  public synchronized void forEach(java.util.function.Consumer<? super E>);
  public synchronized void replaceAll(java.util.function.UnaryOperator<E>);
  public synchronized void sort(java.util.Comparator<? super E>);
  public java.util.Spliterator<E> spliterator();
  void checkInvariants();
}
PS C:\Users\hp>
```

Difference between ArrayList and Vector

ArrayList and Vector both implement List interface and maintain insertion order.

However, there are many differences between ArrayList and Vector classes that are given below.

ArrayList	Vector
1) ArrayList is not synchronized.	Vector is synchronized.
2) ArrayList increments 50% of current array size if the number of elements exceeds its capacity.	Vector increments 100% means double the array size if the total number of elements exceeds its capacity.

3) ArrayList is not a legacy class. It is introduced in JDK 1.2.	**Vector is a legacy class.**
4) ArrayList is fast because it is non-synchronized.	**Vector is slow because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until the current thread releases the lock of the object.**
5) ArrayList uses the Iterator interface to traverse the elements.	**A Vector can use the Iterator interface or Enumeration interface to traverse the elements.**

Stack

- **The stack is the subclass of Vector.**
- **It also implements interfaces List, Collection, Iterable, Cloneable, Serializable.**
- **It implements the last-in-first-out data structure, i.e Stack.**
- **The stack contains all of the methods of the Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.**

We can fetch elements of the stack using three different methods are as follows:

- **Using iterator() Method**
- **Using forEach() Method**

- **Using listIterator() Method**

```
PS C:\Users\hp> javap java.util.Stack
Compiled from "Stack.java"
public class java.util.Stack<E> extends java.util.Vector<E> {
  public java.util.Stack();
  public E push(E);
  public synchronized E pop();
  public synchronized E peek();
  public boolean empty();
  public synchronized int search(java.lang.Object);
}
PS C:\Users\hp>
```

Queue Interface

- **Queue interface maintains the first-in-first-out order.**
- **It can be defined as an ordered list that is used to hold the elements which are about to be processed.**
- **There are various classes like PriorityQueue and ArrayDeque which implement the Queue interface.**
- **PriorityQueue, ArrayBlockingQueue and LinkedList are the implementations that are used most frequently.**
- **The NullPointerException is raised, if any null operation is done on the BlockingQueues.**
- **Those Queues that are present in the *util* package are known as Unbounded Queues.**
- **Those Queues that are present in the *util.concurrent* package are known as bounded Queues.**
- **All Queues barring the Deques facilitate removal and insertion at the head and tail of the queue; respectively. In fact, deques support element insertion and removal at both ends.**
- **Being an interface, the queue requires, for the declaration, a concrete class, and the most common classes are the LinkedList and PriorityQueue in Java.**
- **Implementations done by these classes are not thread safe. If it is required to have a thread safe implementation, PriorityBlockingQueue is an available option.**

Queue interface can be instantiated as:

1. **Queue<String> q1 = new PriorityQueue();**
2. **Queue<String> q2 = new ArrayDeque();**

```
PS C:\Users\hp> javap java.util.Queue
Compiled from "Queue.java"
public interface java.util.Queue<E> extends java.util.Collection<E> {
  public abstract boolean add(E);
  public abstract boolean offer(E);
  public abstract E remove();
  public abstract E poll();
  public abstract E element();
  public abstract E peek();
}
PS C:\Users\hp> _
```

PriorityQueue

- **The PriorityQueue class implements the Queue interface.**
- **It holds the elements or objects which are to be processed by their priorities.**
- **PriorityQueue doesn't allow null values to be stored in the queue.**

```
PS C:\Users\hp> javap java.util.PriorityQueue
Compiled from "PriorityQueue.java"
public class java.util.PriorityQueue<E> extends java.util.AbstractQueue<E> implements java.io.Serializable {
  transient java.lang.Object[] queue;
  int size;
  transient int modCount;
  public java.util.PriorityQueue();
  public java.util.PriorityQueue(int);
  public java.util.PriorityQueue(java.util.Comparator<? super E>);
  public java.util.PriorityQueue(int, java.util.Comparator<? super E>);
  public java.util.PriorityQueue(java.util.Collection<? extends E>);
  public java.util.PriorityQueue(java.util.PriorityQueue<? extends E>);
  public java.util.PriorityQueue(java.util.SortedSet<? extends E>);
  public boolean add(E);
  public boolean offer(E);
  public E peek();
  public boolean remove(java.lang.Object);
  void removeEq(java.lang.Object);
  public boolean contains(java.lang.Object);
  public java.lang.Object[] toArray();
  public <T> T[] toArray(T[]);
  public java.util.Iterator<E> iterator();
  public int size();
  public void clear();
  public E poll();
  E removeAt(int);
  public java.util.Comparator<? super E> comparator();
  public final java.util.Spliterator<E> spliterator();
  public boolean removeIf(java.util.function.Predicate<? super E>);
  public boolean removeAll(java.util.Collection<?>);
  public boolean retainAll(java.util.Collection<?>);
  public void forEach(java.util.function.Consumer<? super E>);
}
PS C:\Users\hp>
```

AbstractQueue

```
PS C:\Users\hp> javap java.util.AbstractQueue
Compiled from "AbstractQueue.java"
public abstract class java.util.AbstractQueue<E> extends java.util.AbstractCollection<E>
  implements java.util.Queue<E> {
  protected java.util.AbstractQueue();
  public boolean add(E);
  public E remove();
  public E element();
  public void clear();
  public boolean addAll(java.util.Collection<? extends E>);
}
PS C:\Users\hp>
```

Deque Interface

- Deque interface extends the Queue interface.
- In Deque, we can remove and add the elements from both sides.
- Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. **Deque d = new ArrayDeque();**

```
PS C:\Users\hp> javap java.util.Deque
Compiled from "Deque.java"
public interface java.util.Deque<E> extends java.util.Queue<E> {
  public abstract void addFirst(E);
  public abstract void addLast(E);
  public abstract boolean offerFirst(E);
  public abstract boolean offerLast(E);
  public abstract E removeFirst();
  public abstract E removeLast();
  public abstract E pollFirst();
  public abstract E pollLast();
  public abstract E getFirst();
  public abstract E getLast();
  public abstract E peekFirst();
  public abstract E peekLast();
  public abstract boolean removeFirstOccurrence(java.lang.Object);
  public abstract boolean removeLastOccurrence(java.lang.Object);
  public abstract boolean add(E);
  public abstract boolean offer(E);
  public abstract E remove();
  public abstract E poll();
  public abstract E element();
  public abstract E peek();
  public abstract boolean addAll(java.util.Collection<? extends E>);
  public abstract void push(E);
  public abstract E pop();
  public abstract boolean remove(java.lang.Object);
  public abstract boolean contains(java.lang.Object);
  public abstract int size();
  public abstract java.util.Iterator<E> iterator();
  public abstract java.util.Iterator<E> descendingIterator();
}
PS C:\Users\hp> _
```


ArrayDeque

- **The ArrayDeque class implements the Deque interface.**

- It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.
- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.

```
PS C:\Users\hp> javap java.util.ArrayDeque
Compiled from "ArrayDeque.java"
public class java.util.ArrayDeque<E> extends java.util.AbstractCollection<E> implements java.util.Deque<E>, jav
a.lang.Cloneable, java.io.Serializable {
  transient java.lang.Object[] elements;
  transient int head;
  transient int tail;
  public java.util.ArrayDeque();
  public java.util.ArrayDeque(int);
  public java.util.ArrayDeque(java.util.Collection<? extends E>);
  static final int inc(int, int);
  static final int dec(int, int);
  static final int inc(int, int, int);
  static final int sub(int, int, int);
  static final <E> E elementAt(java.lang.Object[], int);
  static final <E> E nonNullElementAt(java.lang.Object[], int);
  public void addFirst(E);
  public void addLast(E);
  public boolean addAll(java.util.Collection<? extends E>);
  public boolean offerFirst(E);
  public boolean offerLast(E);
  public E removeFirst();
  public E removeLast();
  public E pollFirst();
  public E pollLast();
  public E getFirst();
  public E getLast();
  public E peekFirst();
  public E peekLast();
  public boolean removeFirstOccurrence(java.lang.Object);
  public boolean removeLastOccurrence(java.lang.Object);
  public boolean add(E);
  public boolean offer(E);
  public E remove();
  public E poll();
  public E element();
  public E peek();
  public void push(E);
  public E pop();
  boolean delete(int);
  public int size();
  public boolean isEmpty();
  public java.util.Iterator<E> iterator();
  public java.util.Iterator<E> descendingIterator();
  public java.util.Spliterator<E> spliterator();
  public void forEach(java.util.function.Consumer<? super E>);
  public boolean removeIf(java.util.function.Predicate<? super E>);
  public boolean removeAll(java.util.Collection<?>);
  public boolean retainAll(java.util.Collection<?>);
  public boolean contains(java.lang.Object);
  public boolean remove(java.lang.Object);
  public void clear();
  public java.lang.Object[] toArray();
  public <T> T[] toArray(T[]);
  public java.util.ArrayDeque<E> clone();
  void checkInvariants();
  public java.lang.Object clone() throws java.lang.CloneNotSupportedException;
}
```


Set Interface

- Set Interface in Java is present in java.util package.
- It extends the Collection interface.
- It represents the unordered set of elements which doesn't allow us to store the duplicate items.
- We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = new HashSet<data-type>();
2. Set<data-type> s2 = new LinkedHashSet<data-type>();
3. Set<data-type> s3 = new TreeSet<data-type>();

```
PS C:\Users\hp> javap java.util.Set
Compiled from "Set.java"
public interface java.util.Set<E> extends java.util.Collection<E> {
  public abstract int size();
  public abstract boolean isEmpty();
  public abstract boolean contains(java.lang.Object);
  public abstract java.util.Iterator<E> iterator();
  public abstract java.lang.Object[] toArray();
  public abstract <T> T[] toArray(T[]);
  public abstract boolean add(E);
  public abstract boolean remove(java.lang.Object);
  public abstract boolean containsAll(java.util.Collection<?>);
  public abstract boolean addAll(java.util.Collection<? extends E>);
  public abstract boolean retainAll(java.util.Collection<?>);
  public abstract boolean removeAll(java.util.Collection<?>);
  public abstract void clear();
  public abstract boolean equals(java.lang.Object);
  public abstract int hashCode();
  public default java.util.Spliterator<E> spliterator();
  public static <E> java.util.Set<E> of();
  public static <E> java.util.Set<E> of(E);
  public static <E> java.util.Set<E> of(E, E);
  public static <E> java.util.Set<E> of(E, E, E);
  public static <E> java.util.Set<E> of(E, E, E, E);
  public static <E> java.util.Set<E> of(E, E, E, E, E);
  public static <E> java.util.Set<E> of(E, E, E, E, E, E);
  public static <E> java.util.Set<E> of(E, E, E, E, E, E, E);
  public static <E> java.util.Set<E> of(E, E, E, E, E, E, E, E);
  public static <E> java.util.Set<E> of(E, E, E, E, E, E, E, E, E);
  public static <E> java.util.Set<E> of(E, E, E, E, E, E, E, E, E, E);
  public static <E> java.util.Set<E> of(E...);
  public static <E> java.util.Set<E> copyOf(java.util.Collection<? extends E>);
}
PS C:\Users\hp>
```


HashSet

- HashSet class inherits the AbstractSet class and implements Set interface.
- It represents the collection that uses a hash table for storage.
- Hashing is used to store the elements in the HashSet.
- It contains unique items.
- HashSet class allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

```
PS C:\Users\hp> javap java.util.HashSet
Compiled from "HashSet.java"
public class java.util.HashSet<E> extends java.util.AbstractSet<E> implements java.util.Set<E>,
java.lang.Cloneable, java.io.Serializable {
  static final long serialVersionUID;
  public java.util.HashSet();
  public java.util.HashSet(java.util.Collection<? extends E>);
  public java.util.HashSet(int, float);
  public java.util.HashSet(int);
  java.util.HashSet(int, float, boolean);
  public java.util.Iterator<E> iterator();
  public int size();
  public boolean isEmpty();
  public boolean contains(java.lang.Object);
  public boolean add(E);
  public boolean remove(java.lang.Object);
  public void clear();
  public java.lang.Object clone();
  public java.util.Spliterator<E> spliterator();
  public java.lang.Object[] toArray();
  public <T> T[] toArray(T[]);
  static {};
}
PS C:\Users\hp>
```

AbstractSet

```
PS C:\Users\hp> javap java.util.AbstractSet
Compiled from "AbstractSet.java"
public abstract class java.util.AbstractSet<E> extends java.util.AbstractCollection<E>
implements java.util.Set<E> {
  protected java.util.AbstractSet();
  public boolean equals(java.lang.Object);
  public int hashCode();
  public boolean removeAll(java.util.Collection<?>);
}
PS C:\Users\hp>
```


LinkedHashSet

- The LinkedHashSet class is a Hashtable and LinkedList implementation of Set Interface.
- It extends the HashSet class and implements Set interface.
- Like HashSet, It also contains unique elements.
- It maintains the insertion order and permits null elements.
- Java LinkedHashSet class is non-synchronized.

```
PS C:\Users\hp> javap java.util.LinkedHashSet
Compiled from "LinkedHashSet.java"
public class java.util.LinkedHashSet<E> extends java.util.HashSet<E> implements java.util.Set<E>
, java.lang.Cloneable, java.io.Serializable {
  public java.util.LinkedHashSet(int, float);
  public java.util.LinkedHashSet(int);
  public java.util.LinkedHashSet();
  public java.util.LinkedHashSet(java.util.Collection<? extends E>);
  public java.util.Spliterator<E> spliterator();
}
PS C:\Users\hp> _
```

SortedSet Interface

- SortedSet is the alternate of Set interface that provides a total ordering on its elements.

- The elements of the SortedSet are arranged in the increasing (ascending) order.
- The SortedSet provides additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

1. SortedSet<data-type> set = new TreeSet();

```
PS C:\Users\hp> javap java.util.SortedSet
Compiled from "SortedSet.java"
public interface java.util.SortedSet<E> extends java.util.Set<E> {
  public abstract java.util.Comparator<? super E> comparator();
  public abstract java.util.SortedSet<E> subSet(E, E);
  public abstract java.util.SortedSet<E> headSet(E);
  public abstract java.util.SortedSet<E> tailSet(E);
  public abstract E first();
  public abstract E last();
  public default java.util.Spliterator<E> spliterator();
}
PS C:\Users\hp>
```

TreeSet

- Java TreeSet class implements the Set interface that uses a tree for storage.
- Like HashSet, TreeSet also contains unique elements.
- However, the access and retrieval time of TreeSet is quite fast.
- The elements in TreeSet are stored in ascending order.
- It inherits AbstractSet class and implements the NavigableSet interface.
- Java TreeSet class doesn't allow null elements.
- Java TreeSet class is non synchronized.
- The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

```
PS C:\Users\hp> javap java.util.TreeSet
Compiled from "TreeSet.java"
public class java.util.TreeSet<E> extends java.util.AbstractSet<E> implements java.util.Navigabl
eSet<E>, java.lang.Cloneable, java.io.Serializable {
  java.util.TreeSet(java.util.NavigableMap<E, java.lang.Object>);
  public java.util.TreeSet();
  public java.util.TreeSet(java.util.Comparator<? super E>);
  public java.util.TreeSet(java.util.Collection<? extends E>);
  public java.util.TreeSet(java.util.SortedSet<E>);
  public java.util.Iterator<E> iterator();
  public java.util.Iterator<E> descendingIterator();
  public java.util.NavigableSet<E> descendingSet();
  public int size();
  public boolean isEmpty();
  public boolean contains(java.lang.Object);
  public boolean add(E);
  public boolean remove(java.lang.Object);
  public void clear();
  public boolean addAll(java.util.Collection<? extends E>);
  public java.util.NavigableSet<E> subSet(E, boolean, E, boolean);
  public java.util.NavigableSet<E> headSet(E, boolean);
  public java.util.NavigableSet<E> tailSet(E, boolean);
  public java.util.SortedSet<E> subSet(E, E);
  public java.util.SortedSet<E> headSet(E);
  public java.util.SortedSet<E> tailSet(E);
  public java.util.Comparator<? super E> comparator();
  public E first();
  public E last();
  public E lower(E);
  public E floor(E);
  public E ceiling(E);
  public E higher(E);
  public E pollFirst();
  public E pollLast();
  public java.lang.Object clone();
  public java.util.Spliterator<E> spliterator();
  static {};
}
PS C:\Users\hp> _
```

NavigableSet

```
PS C:\Users\hp> javap java.util.NavigableSet
Compiled from "NavigableSet.java"
public interface java.util.NavigableSet<E> extends java.util.SortedSet<E> {
  public abstract E lower(E);
  public abstract E floor(E);
  public abstract E ceiling(E);
  public abstract E higher(E);
  public abstract E pollFirst();
  public abstract E pollLast();
  public abstract java.util.Iterator<E> iterator();
  public abstract java.util.NavigableSet<E> descendingSet();
  public abstract java.util.Iterator<E> descendingIterator();
  public abstract java.util.NavigableSet<E> subSet(E, boolean, E, boolean);
  public abstract java.util.NavigableSet<E> headSet(E, boolean);
  public abstract java.util.NavigableSet<E> tailSet(E, boolean);
  public abstract java.util.SortedSet<E> subSet(E, E);
  public abstract java.util.SortedSet<E> headSet(E);
  public abstract java.util.SortedSet<E> tailSet(E);
}
PS C:\Users\hp>
```


Internal Working of The TreeSet Class

TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree. Therefore, operations such as a search, remove, and add consume O(log(N)) time. The reason behind this is there in the self-balancing tree. It is there to ensure that the tree height never exceeds O(log(N)) for all of the mentioned operations. Therefore, it is one of the efficient data structures in order to keep the large data that is sorted and also to do operations on it.

Synchronization of The TreeSet Class

As already mentioned above, the TreeSet class is not synchronized. It means if more than one thread concurrently accesses a tree set, and one of the accessing threads modifies it, then the synchronization must be done manually. It is usually done by doing some object synchronization that encapsulates the set. However, in the case where no such object is found, then the set must be wrapped with the help of the Collections.synchronizedSet() method. It is advised to use the method during creation time in order to avoid the unsynchronized access of the set. The following code snippet shows the same.

```
1. TreeSet treeSet = new TreeSet();
2. Set syncrSet = Collections.synchronziedSet(treeSet);
```

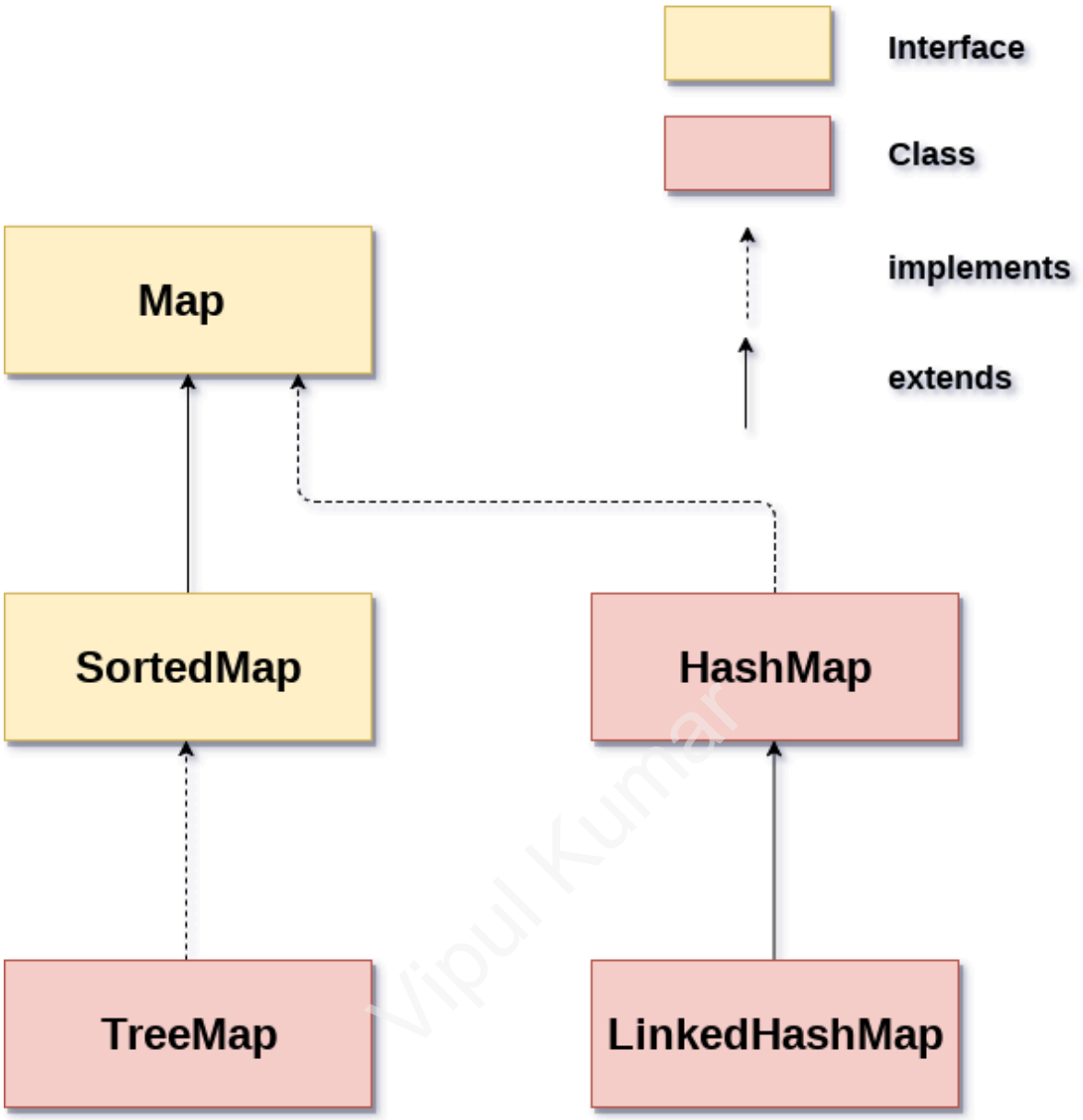
Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



Map.Entry Interface

Entry is the subinterface of Map. So we will be accessing it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

```
PS C:\Users\hp> javap java.util.Map
Compiled from "Map.java"
public interface java.util.Map<K, V> {
  public abstract int size();
  public abstract boolean isEmpty();
  public abstract boolean containsKey(java.lang.Object);
  public abstract boolean containsValue(java.lang.Object);
  public abstract V get(java.lang.Object);
  public abstract V put(K, V);
  public abstract V remove(java.lang.Object);
  public abstract void putAll(java.util.Map<? extends K, ? extends V>);
  public abstract void clear();
  public abstract java.util.Set<K> keySet();
  public abstract java.util.Collection<V> values();
  public abstract java.util.Set<java.util.Map$Entry<K, V>> entrySet();
  public abstract boolean equals(java.lang.Object);
  public abstract int hashCode();
  public default V getOrDefault(java.lang.Object, V);
  public default void forEach(java.util.function.BiConsumer<? super K, ? super V>);
  public default void replaceAll(java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public default V putIfAbsent(K, V);
  public default boolean remove(java.lang.Object, java.lang.Object);
  public default boolean replace(K, V, V);
  public default V replace(K, V);
  public default V computeIfAbsent(K, java.util.function.Function<? super K, ? extends V>);
  public default V computeIfPresent(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public default V compute(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public default V merge(K, V, java.util.function.BiFunction<? super V, ? super V, ? extends V>);
  public static <K, V> java.util.Map<K, V> of();
  public static <K, V> java.util.Map<K, V> of(K, V);
  public static <K, V> java.util.Map<K, V> of(K, V, K, V);
  public static <K, V> java.util.Map<K, V> of(K, V, K, V, K, V);
  public static <K, V> java.util.Map<K, V> of(K, V, K, V, K, V, K, V);
  public static <K, V> java.util.Map<K, V> of(K, V, K, V, K, V, K, V, K, V);
  public static <K, V> java.util.Map<K, V> of(K, V, K, V, K, V, K, V, K, V, K, V);
  public static <K, V> java.util.Map<K, V> of(K, V, K, V, K, V, K, V, K, V, K, V, K, V);
  public static <K, V> java.util.Map<K, V> of(K, V, K, V, K, V, K, V, K, V, K, V, K, V, K, V);
  public static <K, V> java.util.Map<K, V> of(K, V, K, V, K, V, K, V, K, V, K, V, K, V, K, V, K, V);
  public static <K, V> java.util.Map<K, V> ofEntries(java.util.Map$Entry<? extends K, ? extends V>...);
  public static <K, V> java.util.Map$Entry<K, V> entry(K, V);
  public static <K, V> java.util.Map<K, V> copyOf(java.util.Map<? extends K, ? extends V>);
}
PS C:\Users\hp> _
```

```
PS C:\Users\hp> javap java.util.Map.Entry
Compiled from "Map.java"
public interface java.util.Map$Entry<K, V> {
  public abstract K getKey();
  public abstract V getValue();
  public abstract V setValue(V);
  public abstract boolean equals(java.lang.Object);
  public abstract int hashCode();
  public static <K extends java.lang.Comparable<? super K>, V> java.util.Comparator<java.util.Map$Entry<K, V>> comparingByKey();
  public static <K, V extends java.lang.Comparable<? super V>> java.util.Comparator<java.util.Map$Entry<K, V>> comparingByValue();
  public static <K, V> java.util.Comparator<java.util.Map$Entry<K, V>> comparingByKey(java.util.Comparator<? super K>);
  public static <K, V> java.util.Comparator<java.util.Map$Entry<K, V>> comparingByValue(java.util.Comparator<? super V>);
  public static <K, V> java.util.Map$Entry<K, V> copyOf(java.util.Map$Entry<? extends K, ? extends V>);
}
PS C:\Users\hp>
```


HashMap

- **Java HashMap class implements the Map interface which allows us *to store key and value pairs*, where keys should be unique.**
- **HashMap in Java is like the legacy Hashtable class, but it is not synchronized.**
- **It allows us to store the null elements as well, but there should be only one null key.**
- **Java HashMap is non synchronized.**
- **Java HashMap maintains no order.**
- **The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.**

- **HashMap class extends AbstractMap class and implements Map interface.**
- **You cannot store duplicate keys in HashMap. However, if you try to store a duplicate key with another value, it will replace the value.**
- **HashSet contains only values whereas HashMap contains an entry(key and value).**

```
PS C:\Users\hp> javap java.util.HashMap
Compiled from "HashMap.java"
public class java.util.HashMap<K, V> extends java.util.AbstractMap<K, V> implements java.util.Ma
p<K, V>, java.lang.Cloneable, java.io.Serializable {
  static final int DEFAULT_INITIAL_CAPACITY;
  static final int MAXIMUM_CAPACITY;
  static final float DEFAULT_LOAD_FACTOR;
  static final int TREEIFY_THRESHOLD;
  static final int UNTREEIFY_THRESHOLD;
  static final int MIN_TREEIFY_CAPACITY;
  transient java.util.HashMap$Node<K, V>[] table;
  transient java.util.Set<java.util.Map$Entry<K, V>> entrySet;
  transient int size;
  transient int modCount;
  int threshold;
  final float loadFactor;
  static final int hash(java.lang.Object);
  static java.lang.Class<?> comparableClassFor(java.lang.Object);
  static int compareComparables(java.lang.Class<?>, java.lang.Object, java.lang.Object);
  static final int tableSizeFor(int);
  public java.util.HashMap(int, float);
  public java.util.HashMap(int);
  public java.util.HashMap();
  public java.util.HashMap(java.util.Map<? extends K, ? extends V>);
  final void putMapEntries(java.util.Map<? extends K, ? extends V>, boolean);
  public int size();
  public boolean isEmpty();
  public V get(java.lang.Object);
  final java.util.HashMap$Node<K, V> getNode(java.lang.Object);
  public boolean containsKey(java.lang.Object);
  public V put(K, V);
  final V putVal(int, K, V, boolean, boolean);
  final java.util.HashMap$Node<K, V>[] resize();
  final void treeifyBin(java.util.HashMap$Node<K, V>[], int);
  public void putAll(java.util.Map<? extends K, ? extends V>);
  public V remove(java.lang.Object);
  final java.util.HashMap$Node<K, V> removeNode(int, java.lang.Object, java.lang.Object, boolean
, boolean);
  public void clear();
  public boolean containsValue(java.lang.Object);
  public java.util.Set<K> keySet();
  final <T> T[] prepareArray(T[]);
  <T> T[] keysToArray(T[]);
  <T> T[] valuesToArray(T[]);
  public java.util.Collection<V> values();
  public java.util.Set<java.util.Map$Entry<K, V>> entrySet();
  public V getOrDefault(java.lang.Object, V);
  public V putIfAbsent(K, V);
  public boolean remove(java.lang.Object, java.lang.Object);
  public boolean replace(K, V, V);
  public V replace(K, V);
  public V computeIfAbsent(K, java.util.function.Function<? super K, ? extends V>);
  public V computeIfPresent(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>)
;
  public V compute(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public V merge(K, V, java.util.function.BiFunction<? super V, ? super V, ? extends V>);
  public void forEach(java.util.function.BiConsumer<? super K, ? super V>);
  public void replaceAll(java.util.function.BiFunction<? super K, ? super V, ? extends V>);
```

```
  public boolean isEmpty();
  public V get(java.lang.Object);
  final java.util.HashMap$Node<K, V> getNode(java.lang.Object);
  public boolean containsKey(java.lang.Object);
  public V put(K, V);
  final V putVal(int, K, V, boolean, boolean);
  final java.util.HashMap$Node<K, V>[] resize();
  final void treeifyBin(java.util.HashMap$Node<K, V>[], int);
  public void putAll(java.util.Map<? extends K, ? extends V>);
  public V remove(java.lang.Object);
  final java.util.HashMap$Node<K, V> removeNode(int, java.lang.Object, java.lang.Object, boolean
, boolean);
  public void clear();
  public boolean containsValue(java.lang.Object);
  public java.util.Set<K> keySet();
  final <T> T[] prepareArray(T[]);
  <T> T[] keysToArray(T[]);
  <T> T[] valuesToArray(T[]);
  public java.util.Collection<V> values();
  public java.util.Set<java.util.Map$Entry<K, V>> entrySet();
  public V getOrDefault(java.lang.Object, V);
  public V putIfAbsent(K, V);
  public boolean remove(java.lang.Object, java.lang.Object);
  public boolean replace(K, V, V);
  public V replace(K, V);
  public V computeIfAbsent(K, java.util.function.Function<? super K, ? extends V>);
  public V computeIfPresent(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>)
;
  public V compute(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public V merge(K, V, java.util.function.BiFunction<? super V, ? super V, ? extends V>);
  public void forEach(java.util.function.BiConsumer<? super K, ? super V>);
  public void replaceAll(java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public java.lang.Object clone();
  final float loadFactor();
  final int capacity();
  java.util.HashMap$Node<K, V> newNode(int, K, V, java.util.HashMap$Node<K, V>);
  java.util.HashMap$Node<K, V> replacementNode(java.util.HashMap$Node<K, V>, java.util.HashMap$N
ode<K, V>);
  java.util.HashMap$TreeNode<K, V> newTreeNode(int, K, V, java.util.HashMap$Node<K, V>);
  java.util.HashMap$TreeNode<K, V> replacementTreeNode(java.util.HashMap$Node<K, V>, java.util.H
ashMap$Node<K, V>);
  void reinitialize();
  void afterNodeAccess(java.util.HashMap$Node<K, V>);
  void afterNodeInsertion(boolean);
  void afterNodeRemoval(java.util.HashMap$Node<K, V>);
  void internalWriteEntries(java.io.ObjectOutputStream) throws java.io.IOException;
}
PS C:\Users\hp>
```


AbstractMap

```
PS C:\Users\hp> javap java.util.AbstractMap
Compiled from "AbstractMap.java"
public abstract class java.util.AbstractMap<K, V> implements java.util.Map<K, V> {
  transient java.util.Set<K> keySet;
  transient java.util.Collection<V> values;
  protected java.util.AbstractMap();
  public int size();
  public boolean isEmpty();
  public boolean containsValue(java.lang.Object);
  public boolean containsKey(java.lang.Object);
  public V get(java.lang.Object);
  public V put(K, V);
  public V remove(java.lang.Object);
  public void putAll(java.util.Map<? extends K, ? extends V>);
  public void clear();
  public java.util.Set<K> keySet();
  public java.util.Collection<V> values();
  public abstract java.util.Set<java.util.Map$Entry<K, V>> entrySet();
  public boolean equals(java.lang.Object);
  public int hashCode();
  public java.lang.String toString();
  protected java.lang.Object clone() throws java.lang.CloneNotSupportedException;
}
PS C:\Users\hp> _
```

Working of HashMap

HashMap is a part of the Java collection framework.

It uses a technique called Hashing(It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.).

It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

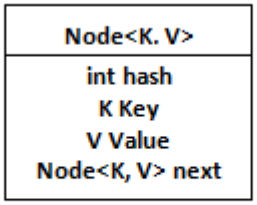


Figure: Representation of a Node

- **equals(): It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.**
- **hashCode(): This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.**
- **Buckets: Arrays of the node are called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.**

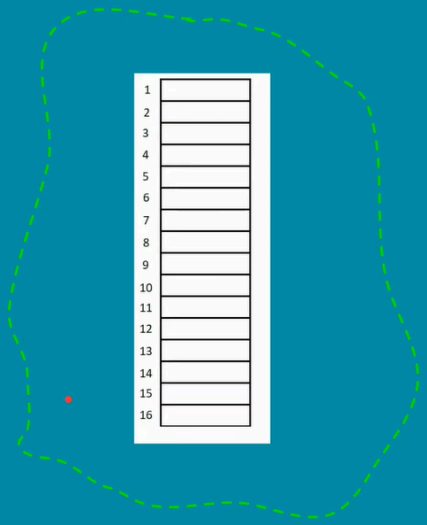
1. Internal structure or working of Hashmap.

What internally happens when, JVM creates new HashMap

```
HashMap<String, String> map = new HashMap<>();
```


1. Internal structure or working of Hashmap.

What internally happens when, JVM creates new HashMap




```
HashMap<String, String> map = new HashMap<>();
```

1. Internal structure or working of Hashmap.

only 16 buckets..??

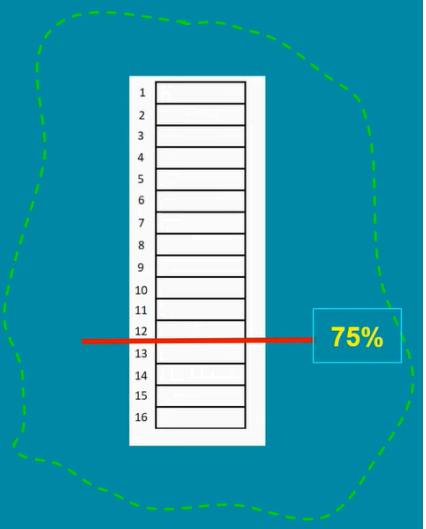
So what if I want to put more than 16 items..??

Load factor => 75% or 0.75 or 3/4th

CONCEPT

Load Factor

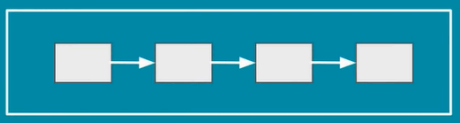
If HashMap reaches more than 75% of it's capacity then it doubles the existing capacity



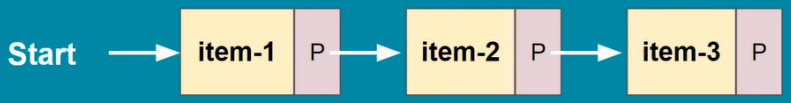
1. Internal structure or working of Hashmap.

Before diving into the internals, let's see what exactly inside a bucket.!!

Linked List

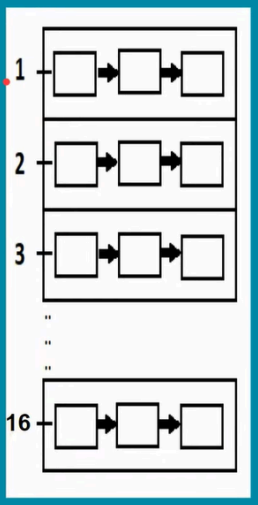


Let's see what happens when we ADD nodes to Linked List



1. Internal structure or working of Hashmap.

Now we know How exactly buckets of HashMap look like



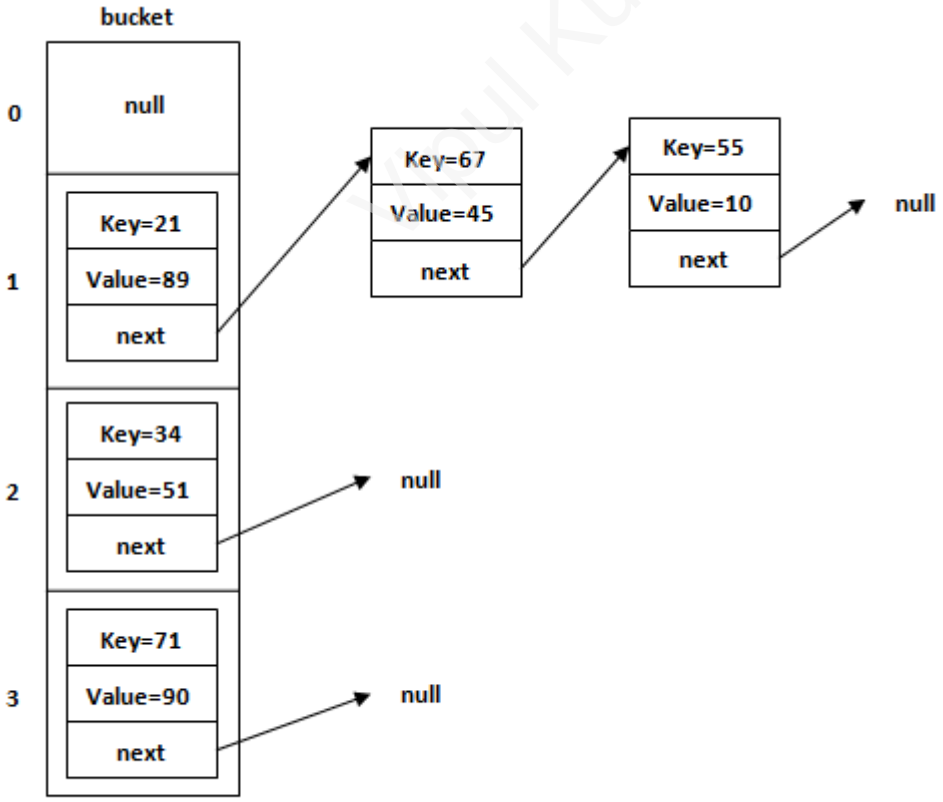
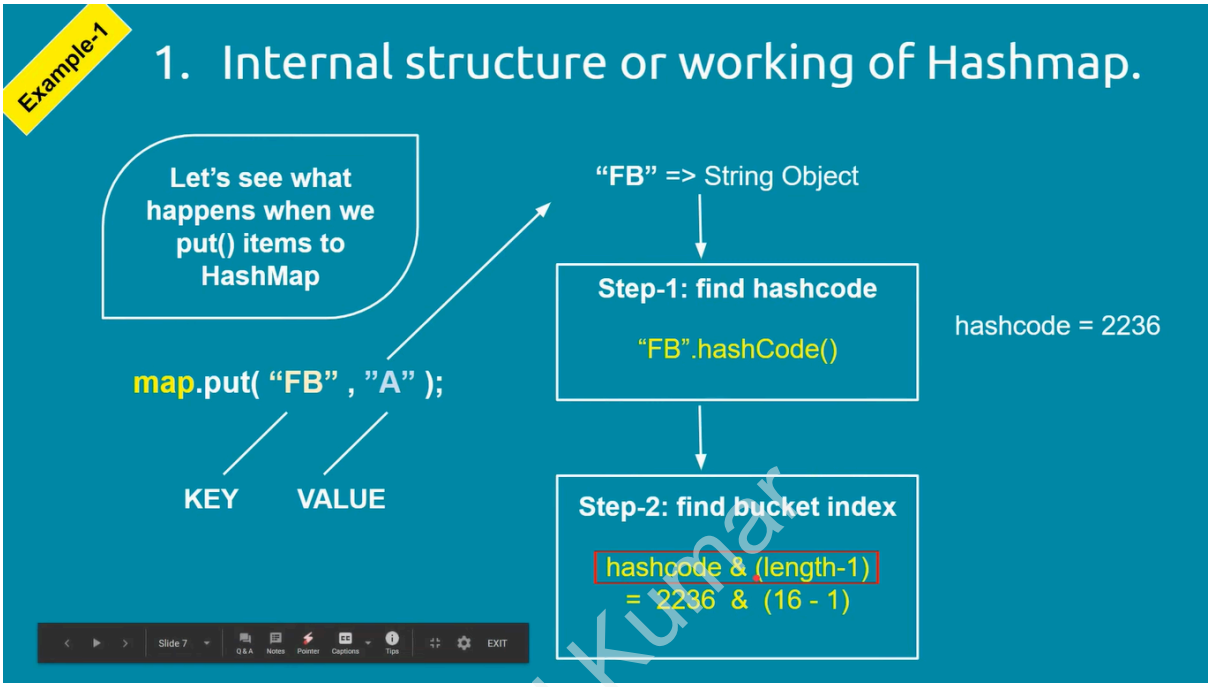
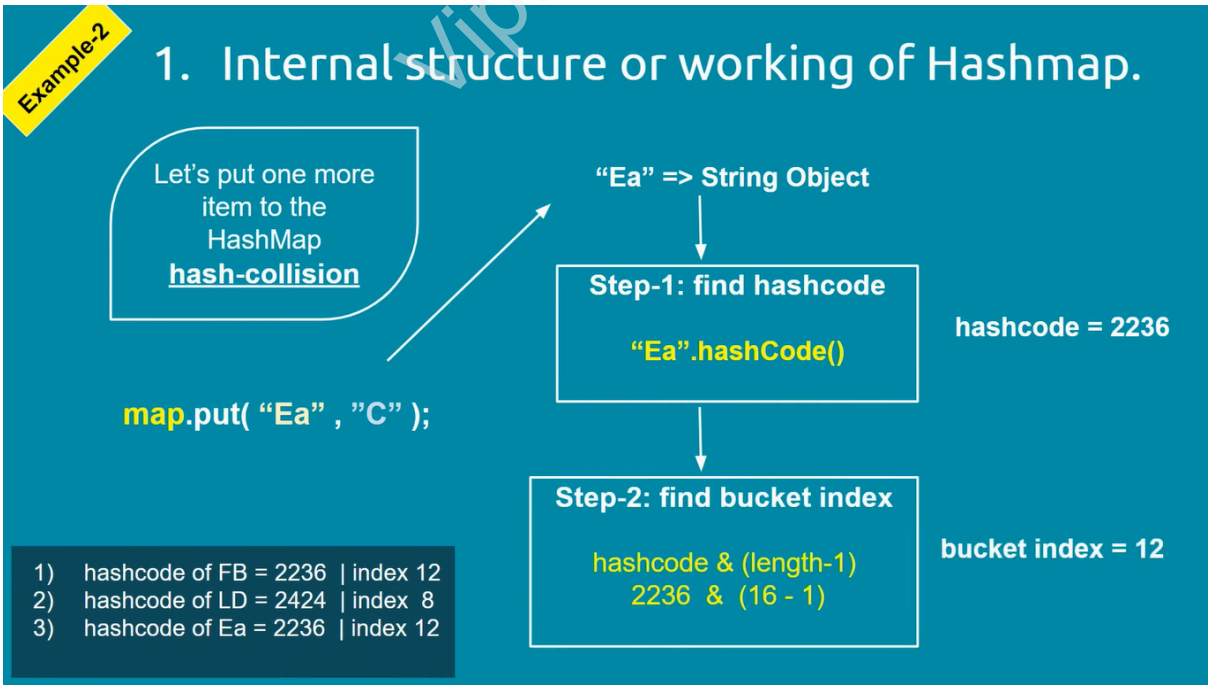


Figure: Allocation of nodes in Bucket

We use the put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

BucketIndex = hashcode(Key) & (n-1)

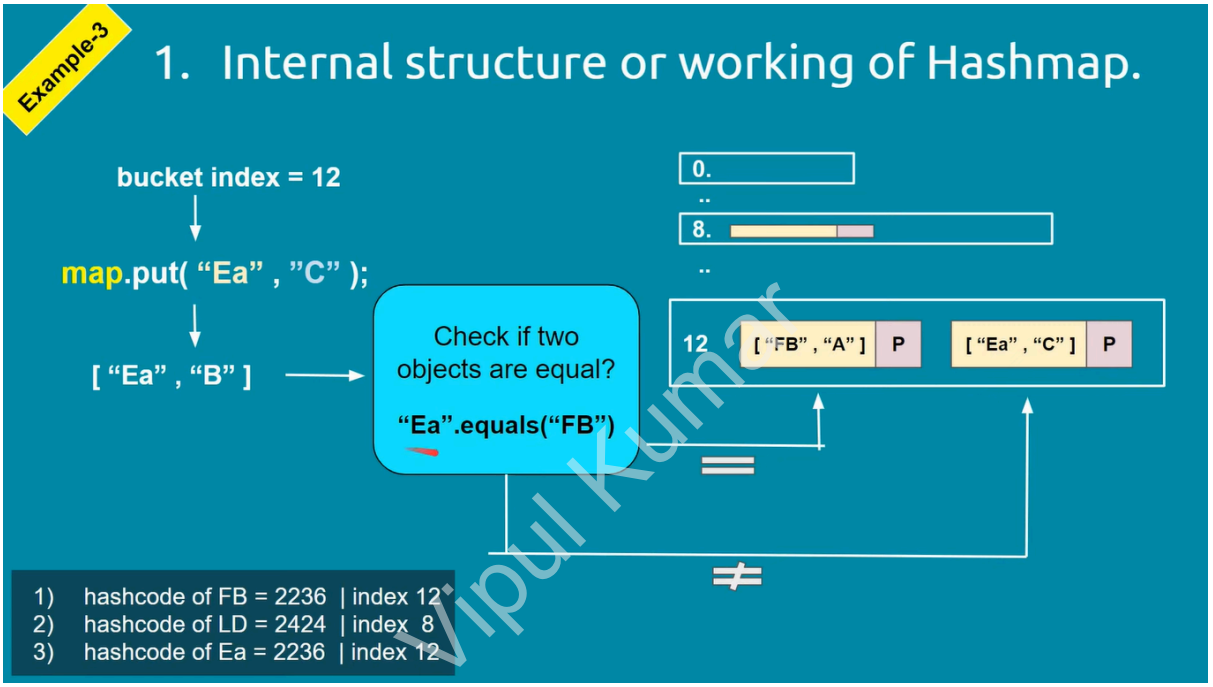


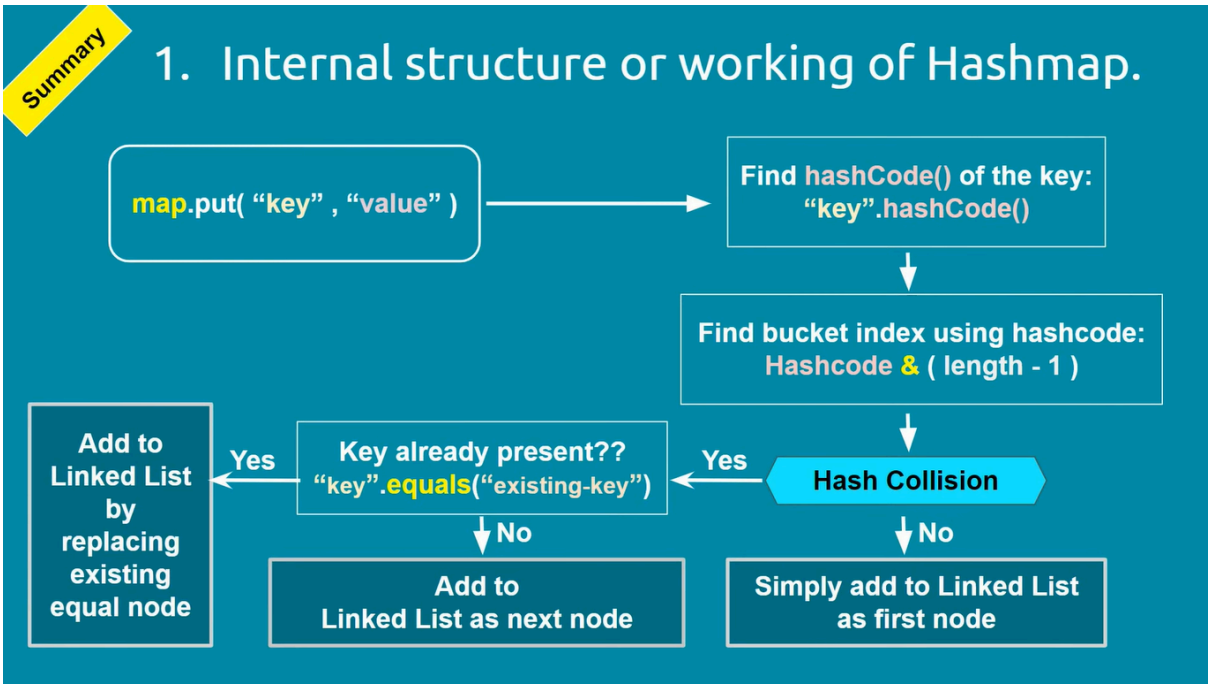


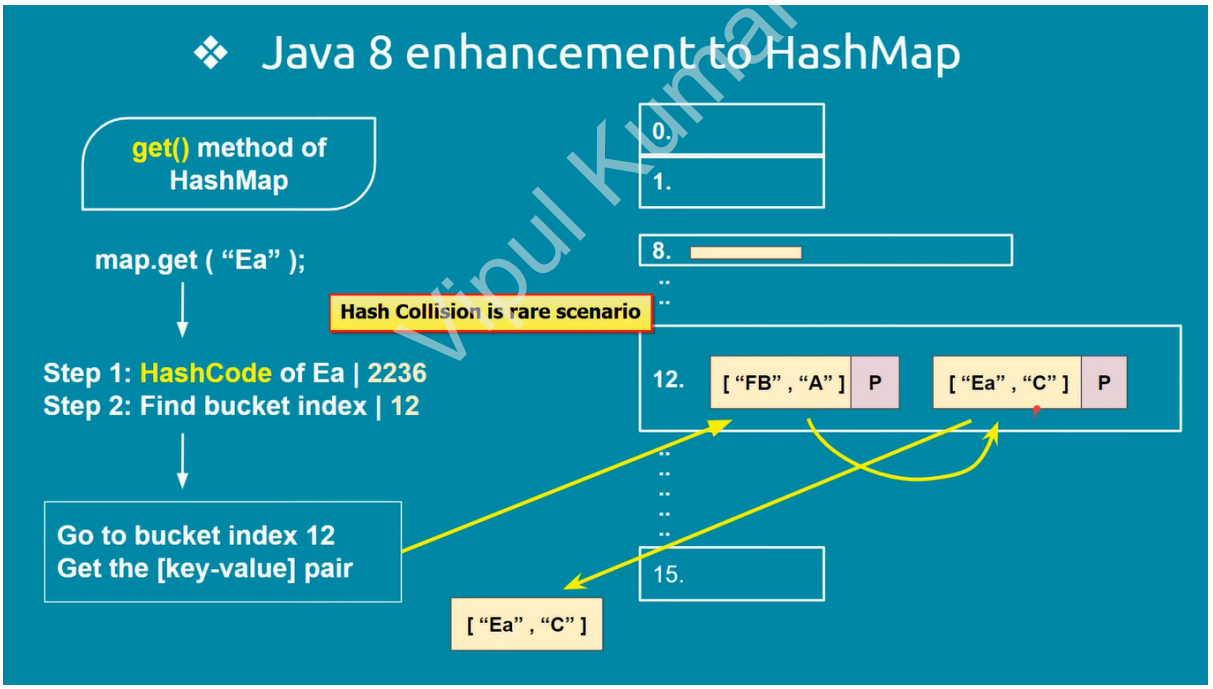
Hash Collision

This is the case when the calculated index value is the same for two or more Keys.

The value 12 is the computed index value where the Key will be stored in HashMap(See fig.). In this case, the equals() method checks if both Keys are equal or not. If Keys are the same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 12.



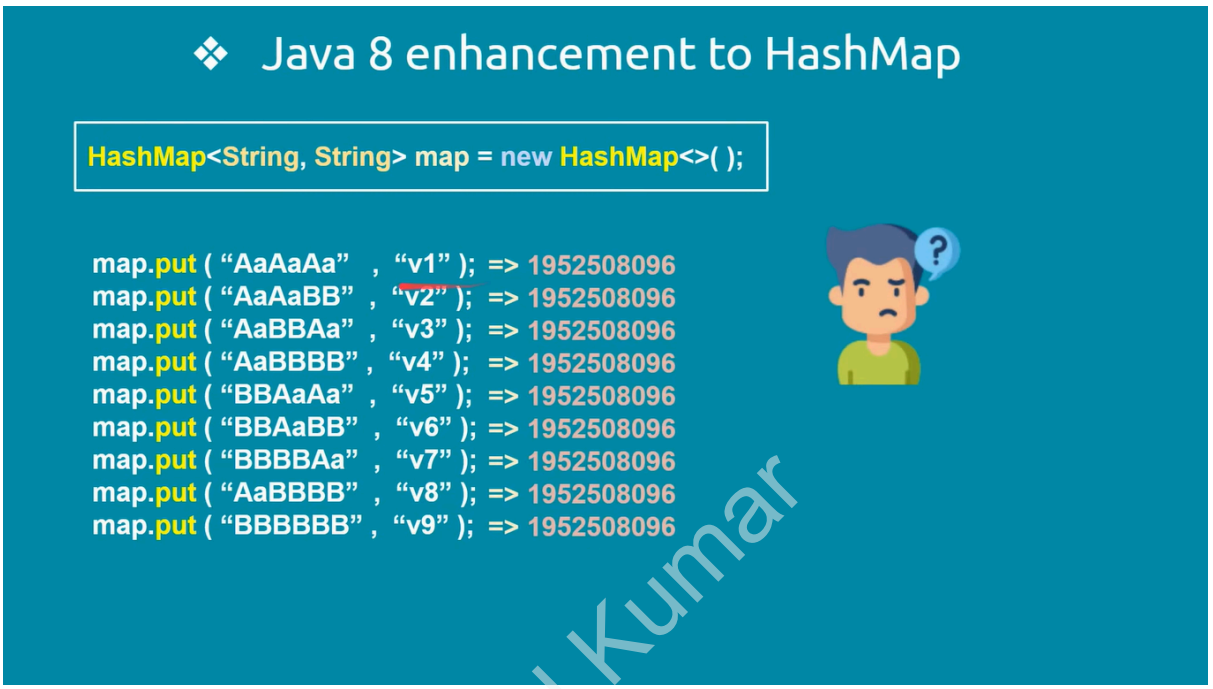


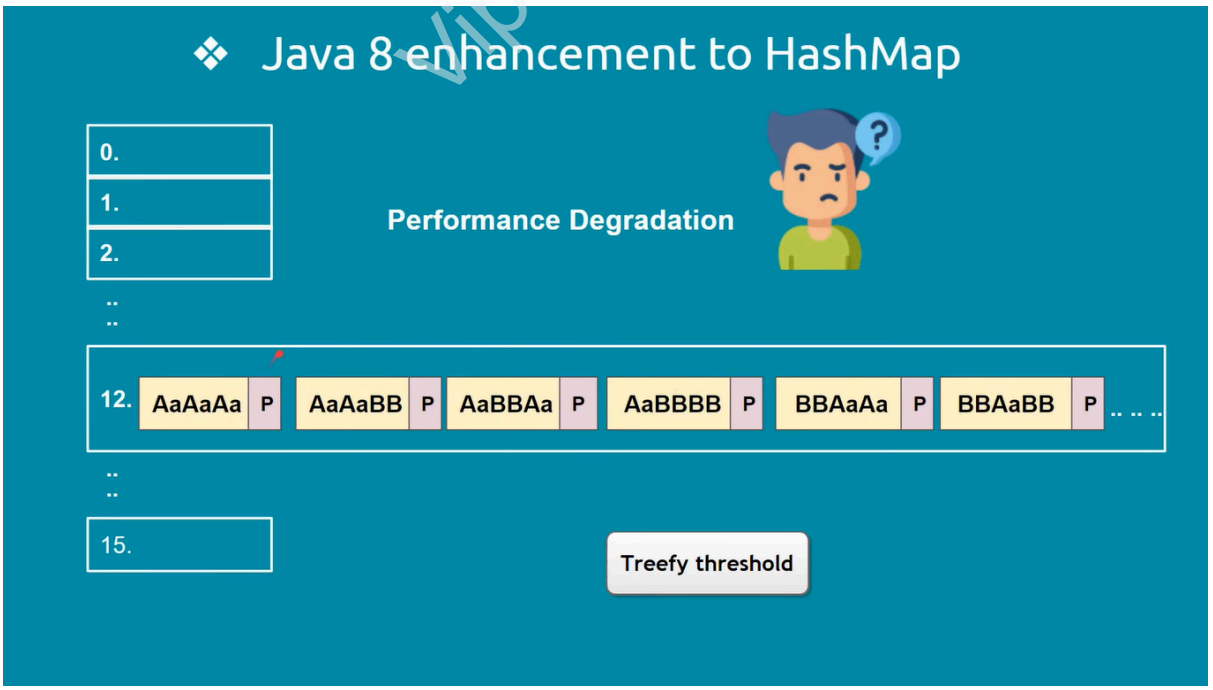


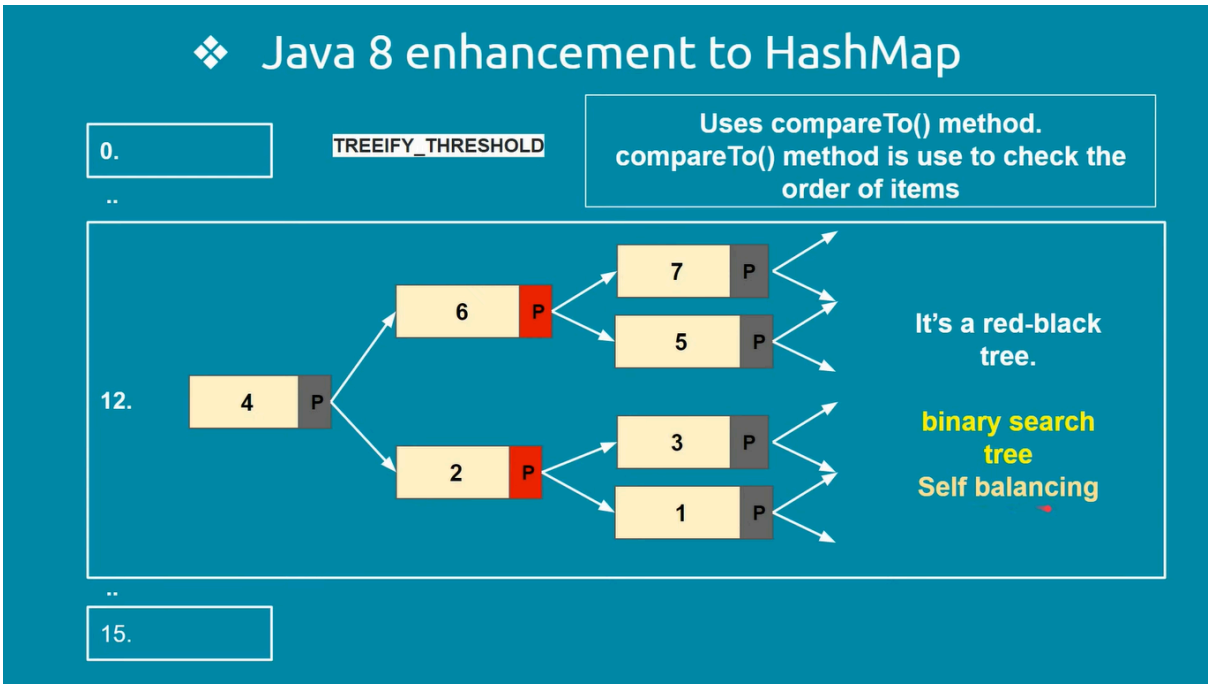
get() method is used to get the value by its Key. When the get(K Key) method is called, it calculates the hash code of the Key. Then it generates an index by using the index formula (same index formula i.e used put() method). Suppose the index generated is 12. Then get() method search for the bucket index value 12. It compares the first element Key with the given Key. If both keys are equal, then it returns the value and if not then checks for the next element in the node if it exists.

In our scenario, it is found as the second element of the node and returns the value C.

TREEIFY_THRESHOLD







Now, suppose so many keys generate the same hashcode. Then in this case the bucket index will be the same and all entries will be put into the same bucket index as shown in the figure. Due to this there will be performance degradation while doing any operation on hashmap.

To resolve this issue the *TREEIFY_THRESHOLD* concept is given.

HashMap initially uses LinkedList. Then when the number of entries crosses a certain threshold, it will replace a *LinkedList* with a balanced binary tree.

Java 8 and later versions use a balanced tree (also called a red-black tree) instead of a *LinkedList* to store collided entries. This improves the worst-case performance of *HashMap* from *O(n)* to *O(log n)*.

The *TREEIFY_THRESHOLD* constant decides this threshold value. Currently, this value is 8, which means if there are more than 8 elements in the same bucket, *Map* will use a tree to hold them.

LinkedHashMap

- **The LinkedHashMap class is a Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.**
- **Java LinkedHashMap contains values based on the key.**
- **Java LinkedHashMap contains unique elements.**
- **Java LinkedHashMap may have one null key and multiple null values.**
- **Java LinkedHashMap is non synchronized.**
- **Java LinkedHashMap maintains insertion order.**
- **The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.**

```
PS C:\Users\hp> javap java.util.LinkedHashMap
Compiled from "LinkedHashMap.java"
public class java.util.LinkedHashMap<K, V> extends java.util.HashMap<K, V> implements java.util.Map<K, V> {
  transient java.util.LinkedHashMap$Entry<K, V> head;
  transient java.util.LinkedHashMap$Entry<K, V> tail;
  final boolean accessOrder;
  void reinitialize();
  java.util.HashMap$Node<K, V> newNode(int, K, V, java.util.HashMap$Node<K, V>);
  java.util.HashMap$Node<K, V> replacementNode(java.util.HashMap$Node<K, V>, java.util.HashMap$Node<K, V>);
  java.util.HashMap$TreeNode<K, V> newTreeNode(int, K, V, java.util.HashMap$Node<K, V>);
  java.util.HashMap$TreeNode<K, V> replacementTreeNode(java.util.HashMap$Node<K, V>, java.util.HashMap$Node<K, V>);
  void afterNodeRemoval(java.util.HashMap$Node<K, V>);
  void afterNodeInsertion(boolean);
  void afterNodeAccess(java.util.HashMap$Node<K, V>);
  void internalWriteEntries(java.io.ObjectOutputStream) throws java.io.IOException;
  public java.util.LinkedHashMap(int, float);
  public java.util.LinkedHashMap(int);
  public java.util.LinkedHashMap();
  public java.util.LinkedHashMap(java.util.Map<? extends K, ? extends V>);
  public java.util.LinkedHashMap(int, float, boolean);
  public boolean containsValue(java.lang.Object);
  public V get(java.lang.Object);
  public V getOrDefault(java.lang.Object, V);
  public void clear();
  protected boolean removeEldestEntry(java.util.Map$Entry<K, V>);
  public java.util.Set<K> keySet();
  final <T> T[] keysToArray(T[]);
  final <T> T[] valuesToArray(T[]);
  public java.util.Collection<V> values();
  public java.util.Set<java.util.Map$Entry<K, V>> entrySet();
  public void forEach(java.util.function.BiConsumer<? super K, ? super V>);
  public void replaceAll(java.util.function.BiFunction<? super K, ? super V, ? extends V>);
}
PS C:\Users\hp>
```


TreeMap

- **TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.**
- **Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.**
- **Java TreeMap contains only unique elements.**
- **Java TreeMap cannot have a null key but can have multiple null values.**
- **Java TreeMap is non synchronized.**
- **Java TreeMap maintains ascending order.**

```
PS C:\Users\hp> javap java.util.TreeMap
Compiled from "TreeMap.java"
public class java.util.TreeMap<K, V> extends java.util.AbstractMap<K, V> implements java.util.Nav
igableMap<K, V>, java.lang.Cloneable, java.io.Serializable {
  public java.util.TreeMap();
  public java.util.TreeMap(java.util.Comparator<? super K>);
  public java.util.TreeMap(java.util.Map<? extends K, ? extends V>);
  public java.util.TreeMap(java.util.SortedMap<K, ? extends V>);
  public int size();
  public boolean containsKey(java.lang.Object);
  public boolean containsValue(java.lang.Object);
  public V get(java.lang.Object);
  public java.util.Comparator<? super K> comparator();
  public K firstKey();
  public K lastKey();
  public void putAll(java.util.Map<? extends K, ? extends V>);
  final java.util.TreeMap$Entry<K, V> getEntry(java.lang.Object);
  final java.util.TreeMap$Entry<K, V> getEntryUsingComparator(java.lang.Object);
  final java.util.TreeMap$Entry<K, V> getCeilingEntry(K);
  final java.util.TreeMap$Entry<K, V> getFloorEntry(K);
  final java.util.TreeMap$Entry<K, V> getHigherEntry(K);
  final java.util.TreeMap$Entry<K, V> getLowerEntry(K);
  public V put(K, V);
  public V putIfAbsent(K, V);
  public V computeIfAbsent(K, java.util.function.Function<? super K, ? extends V>);
  public V computeIfPresent(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);

  public V compute(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public V merge(K, V, java.util.function.BiFunction<? super V, ? super V, ? extends V>);
  public V remove(java.lang.Object);
  public void clear();
  public java.lang.Object clone();
  public java.util.Map$Entry<K, V> firstEntry();
  public java.util.Map$Entry<K, V> lastEntry();
  public java.util.Map$Entry<K, V> pollFirstEntry();
  public java.util.Map$Entry<K, V> pollLastEntry();
  public java.util.Map$Entry<K, V> lowerEntry(K);
  public K lowerKey(K);
  public java.util.Map$Entry<K, V> floorEntry(K);
  public K floorKey(K);
  public java.util.Map$Entry<K, V> ceilingEntry(K);
  public K ceilingKey(K);
  public java.util.Map$Entry<K, V> higherEntry(K);
  public K higherKey(K);
  public java.util.Set<K> keySet();
  public java.util.NavigableSet<K> navigableKeySet();
  public java.util.NavigableSet<K> descendingKeySet();
  public java.util.Collection<V> values();
  public java.util.Set<java.util.Map$Entry<K, V>> entrySet();
```



```
  public java.util.Map$Entry<K, V> ceilingEntry(K);
  public K ceilingKey(K);
  public java.util.Map$Entry<K, V> higherEntry(K);
  public K higherKey(K);
  public java.util.Set<K> keySet();
  public java.util.NavigableSet<K> navigableKeySet();
  public java.util.NavigableSet<K> descendingKeySet();
  public java.util.Collection<V> values();
  public java.util.Set<java.util.Map$Entry<K, V>> entrySet();
  public java.util.NavigableMap<K, V> descendingMap();
  public java.util.NavigableMap<K, V> subMap(K, boolean, K, boolean);
  public java.util.NavigableMap<K, V> headMap(K, boolean);
  public java.util.NavigableMap<K, V> tailMap(K, boolean);
  public java.util.SortedMap<K, V> subMap(K, K);
  public java.util.SortedMap<K, V> headMap(K);
  public java.util.SortedMap<K, V> tailMap(K);
  public boolean replace(K, V, V);
  public V replace(K, V);
  public void forEach(java.util.function.BiConsumer<? super K, ? super V>);
  public void replaceAll(java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  java.util.Iterator<K> keyIterator();
  java.util.Iterator<K> descendingKeyIterator();
  final int compare(java.lang.Object, java.lang.Object);
  static final boolean valEquals(java.lang.Object, java.lang.Object);
  static <K, V> java.util.Map$Entry<K, V> exportEntry(java.util.TreeMap$Entry<K, V>);
  static <K, V> K keyOrNull(java.util.TreeMap$Entry<K, V>);
  static <K> K key(java.util.TreeMap$Entry<K, ?>);
  final java.util.TreeMap$Entry<K, V> getFirstEntry();
  final java.util.TreeMap$Entry<K, V> getLastEntry();
  static <K, V> java.util.TreeMap$Entry<K, V> successor(java.util.TreeMap$Entry<K, V>);
  static <K, V> java.util.TreeMap$Entry<K, V> predecessor(java.util.TreeMap$Entry<K, V>);
  void readTreeSet(int, java.io.ObjectInputStream, V) throws java.io.IOException, java.lang.Class
NotFoundException;
  void addAllForTreeSet(java.util.SortedSet<? extends K>, V);
  static <K> java.util.Spliterator<K> keySpliteratorFor(java.util.NavigableMap<K, ?>);
  final java.util.Spliterator<K> keySpliterator();
  final java.util.Spliterator<K> descendingKeySpliterator();
  static {};

PS C:\Users\hp>
```

NavigableMap

```
PS C:\Users\hp> javap java.util.NavigableMap
Compiled from "NavigableMap.java"
public interface java.util.NavigableMap<K, V> extends java.util.SortedMap<K, V> {
  public abstract java.util.Map$Entry<K, V> lowerEntry(K);
  public abstract K lowerKey(K);
  public abstract java.util.Map$Entry<K, V> floorEntry(K);
  public abstract K floorKey(K);
  public abstract java.util.Map$Entry<K, V> ceilingEntry(K);
  public abstract K ceilingKey(K);
  public abstract java.util.Map$Entry<K, V> higherEntry(K);
  public abstract K higherKey(K);
  public abstract java.util.Map$Entry<K, V> firstEntry();
  public abstract java.util.Map$Entry<K, V> lastEntry();
  public abstract java.util.Map$Entry<K, V> pollFirstEntry();
  public abstract java.util.Map$Entry<K, V> pollLastEntry();
  public abstract java.util.NavigableMap<K, V> descendingMap();
  public abstract java.util.NavigableSet<K> navigableKeySet();
  public abstract java.util.NavigableSet<K> descendingKeySet();
  public abstract java.util.NavigableMap<K, V> subMap(K, boolean, K, boolean);
  public abstract java.util.NavigableMap<K, V> headMap(K, boolean);
  public abstract java.util.NavigableMap<K, V> tailMap(K, boolean);
  public abstract java.util.SortedMap<K, V> subMap(K, K);
  public abstract java.util.SortedMap<K, V> headMap(K);
  public abstract java.util.SortedMap<K, V> tailMap(K);
}
PS C:\Users\hp>
```

SortedMap

```
PS C:\Users\hp> javap java.util.SortedMap
Compiled from "SortedMap.java"
public interface java.util.SortedMap<K, V> extends java.util.Map<K, V> {
  public abstract java.util.Comparator<? super K> comparator();
  public abstract java.util.SortedMap<K, V> subMap(K, K);
  public abstract java.util.SortedMap<K, V> headMap(K);
  public abstract java.util.SortedMap<K, V> tailMap(K);
  public abstract K firstKey();
  public abstract K lastKey();
  public abstract java.util.Set<K> keySet();
  public abstract java.util.Collection<V> values();
  public abstract java.util.Set<java.util.Map$Entry<K, V>> entrySet();
}
PS C:\Users\hp>
```


Hashtable

- **Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.**
- **A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.**
- **Java Hashtable class contains unique elements.**
- **Java Hashtable class doesn't allow null key or value.**
- **Java Hashtable class is synchronized.**
- **The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.**

```
PS C:\Users\hp> javap java.util.Hashtable
Compiled from "Hashtable.java"
public class java.util.Hashtable<K, V> extends java.util.Dictionary<K, V> implements java.util.Map<K, V>, java.lang.Cloneable, java.io.Serializable {
  public java.util.Hashtable(int, float);
  public java.util.Hashtable(int);
  public java.util.Hashtable();
  public java.util.Hashtable(java.util.Map<? extends K, ? extends V>);
  java.util.Hashtable(java.lang.Void);
  public synchronized int size();
  public synchronized boolean isEmpty();
  public synchronized java.util.Enumeration<K> keys();
  public synchronized java.util.Enumeration<V> elements();
  public synchronized boolean contains(java.lang.Object);
  public boolean containsValue(java.lang.Object);
  public synchronized boolean containsKey(java.lang.Object);
  public synchronized V get(java.lang.Object);
  protected void rehash();
  public synchronized V put(K, V);
  public synchronized V remove(java.lang.Object);
  public synchronized void putAll(java.util.Map<? extends K, ? extends V>);
  public synchronized void clear();
  public synchronized java.lang.Object clone();
  final java.util.Hashtable<?, ?> cloneHashtable();
  public synchronized java.lang.String toString();
  public java.util.Set<K> keySet();
  public java.util.Set<java.util.Map$Entry<K, V>> entrySet();
  public java.util.Collection<V> values();
  public synchronized boolean equals(java.lang.Object);
  public synchronized int hashCode();
  public synchronized V getOrDefault(java.lang.Object, V);
  public synchronized void forEach(java.util.function.BiConsumer<? super K, ? super V>);
  public synchronized void replaceAll(java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public synchronized V putIfAbsent(K, V);
  public synchronized boolean remove(java.lang.Object, java.lang.Object);
  public synchronized boolean replace(K, V, V);
  public synchronized V replace(K, V);
  public synchronized V computeIfAbsent(K, java.util.function.Function<? super K, ? extends V>);
  public synchronized V computeIfPresent(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public synchronized V compute(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public synchronized V merge(K, V, java.util.function.BiFunction<? super V, ? super V, ? extends V>);
  void writeHashtable(java.io.ObjectOutputStream) throws java.io.IOException;
  final void defaultWriteHashtable(java.io.ObjectOutputStream, int, float) throws java.io.IOException;
  void readHashtable(java.io.ObjectInputStream) throws java.io.IOException, java.lang.ClassNotFoundException;
}
PS C:\Users\hp>
```

```
PS C:\Users\hp> javap java.util.Dictionary
Compiled from "Dictionary.java"
public abstract class java.util.Dictionary<K, V> {
  public java.util.Dictionary();
  public abstract int size();
  public abstract boolean isEmpty();
  public abstract java.util.Enumeration<K> keys();
  public abstract java.util.Enumeration<V> elements();
  public abstract V get(java.lang.Object);
  public abstract V put(K, V);
  public abstract V remove(java.lang.Object);
}
PS C:\Users\hp>
```


Difference between HashMap and Hashtable

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing techniques to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

HashMap	Hashtable

1) HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is synchronized. It is thread-safe and can be shared with many threads.
2) HashMap allows one null key and multiple null values.	Hashtable doesn't allow any null key or value.
3) HashMap is a new class introduced in JDK 1.2.	Hashtable is a legacy class.
4) HashMap is fast.	Hashtable is slow.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
6) HashMap is traversed by Iterator.	Hashtable is traversed by Enumerator and Iterator.
7) Iterators in HashMap are fail-fast.	Enumerator in Hashtable is not fail-fast.
8) HashMap inherits AbstractMap class.	Hashtable inherits Dictionary class.

Fail-fast and Fail-safe iterations

Using iterations we can traverse over the collections objects. The iterators can be either fail-safe or fail-fast.
***Fail-safe* iterators means they will not throw any exception even if the collection is modified while iterating over it.**
Whereas *Fail-fast* iterators throw an exception(*ConcurrentModificationException*) if the collection is modified while iterating over it.
Consider an example:

```
ArrayList<Integer> integers = new ArrayList<>();
integers.add(1);
integers.add(2);
integers.add(3);
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
  Integer a = itr.next();
  integers.remove(a);
}
```

As arrayLists are fail-fast above code will throw an exception.
First a will have value = 1, and then 1 will be removed in the same iteration.
Next when a will try to get next(), as the modification is made to the list, it will throw an exception here.
However if we use an fail-safe collection e.g. CopyOnWriteArrayList then no exception will occur:

```
List<Integer> integers = new CopyOnWriteArrayList<>();
integers.add(1);
integers.add(2);
integers.add(3);
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
  Integer a = itr.next();
  integers.remove(a);
}
```

Here if we print the element a, then all the elements will be printed.

Fail-Fast Iterators internal working:

Every fail fast collection has a modCount field, to represent how many times the collection has changed/modified.
So at every modification of this collection we increment the modCount value. For example the modCount is incremented in below cases:
1. When one or more elements are removed.
2. When one or more elements are added.
3. When the collection is replaced with another collection.
4. When the collection is sorted.
So everytime there is some change in the collection structure, the mod count is incremented.
Now the iterator stores the modCount value in the initialization as below:
`int expectedModCount = modCount;`
Now while the iteration is going on, expectedModCount will have the old value of modCount.
If there is any change made in the collection, the modCOunt will change and then an exception is thrown using:

```
if (modCount != expectedModCount)
  throw new ConcurrentModificationException();
```

This code is used in most of the iterator methods e.g.

1. next()

2. remove()

3. add()

So if we make any changes to the collection, the modCount will change, and expectedModCount will not be hence equal to the modCount. Then if we use any of the above methods of iterator, the ConcurrentModificationException will be thrown.

Note: If we remove/add the element using the remove() or add() of iterator instead of collection, then in that case no exception will opccur. It is because the remove/add methods of iterators call the remove/add method of collection internally, and also it reasigns the expectedModCount to new modCount value.

```
ArrayList.this.remove(lastRet);
```

```
cursor = lastRet;


lastRet = -1;
expectedModCount = modCount;
and
ArrayList.this.add(i, e);
expectedModCount = modCount;
```

So the below code is safe as we are removing the element from the iterator here:

```
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
  if (itr.next() == 2) {
     // will not throw Exception
     itr.remove();
  }
}
```

Whereas the below code will throw an exception as we are removing the element

from the collection here:

```
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
  if (itr.next() == 3) {
     // will throw Exception on
     // next call of next() method
     integers.remove(3);
  }
}
```

Fail-Safe Iterators internal working:

Unlike the fail-fast iterators, these iterators traverse over the clone of the collection. So even if the original collection gets structurally modified, no exception will be thrown.
E.g. in case of CopyOnWriteArrayList the original collections is passed and is stored in the iterator:

```
public Iterator<E> iterator() {
```

```
  return new COWIterator<E>(getArray(), 0);
}
```

here iterator() method returns the iterator of the CopyOnWriteArrayList. As we can see, it passes the getArray() in the constructor of the iterator. This getArray() has all the collection elements.

Now the iterator(COWIterator here) will save this to traverse upon as:

```
COWIterator(Object[] elements, int initialCursor) {
  cursor = initialCursor;
  snapshot = elements;
}
```

So the original collection elements are saved in the snapshot field variable.

So all the iterator methods will work on this snapshot method. So even if there is any change in the original collection, no exception will be thrown. But note the the iterator will not reflect the latest state of the collection.

```
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
  int a = itr.next();
  if (a == 1) {
     integers.remove(Integer.valueOf(a));
  }
  System.out.print(a);
}
```

So we are removing the element from the collection. the collection now has only elements 2 and 3 in it. But the iterator will print all the elements 1,2,3 because it traverses over the snapshot of the collection elements.

We can print the collection elements after the above code. It will print only 2,3 as:

```
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
  int a = itr.next();
```

```
    System.out.print(a);
  }
```

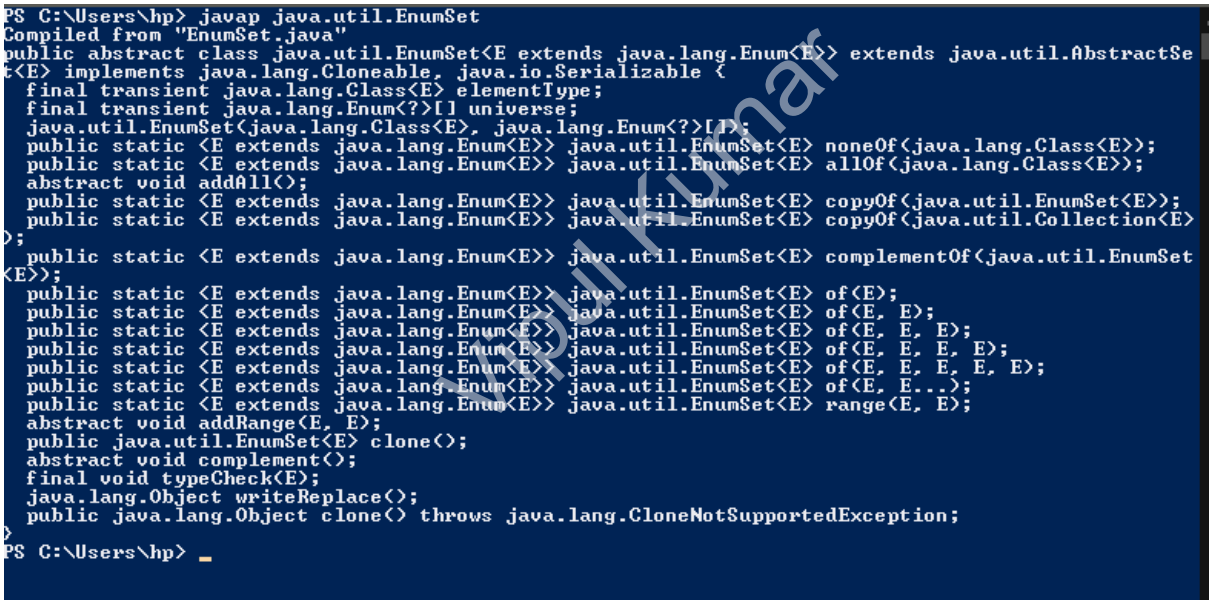
Note: although it does not throw any exception, but the downsides of this iterator are:

1. They will not reflect the latest state of the collection.

2. It requires extra memory as it clones the collection.

EnumSet

EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.




```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
```

```
    // Traversing elements
    Iterator<days> iter = set.iterator();
    while (iter.hasNext())
      System.out.println(iter.next());
  }
}

import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set1 = EnumSet.allOf(days.class);
      System.out.println("Week Days:"+set1);
      Set<days> set2 = EnumSet.noneOf(days.class);
      System.out.println("Week Days:"+set2);
  }
}
```

EnumMap

EnumMap class is the specialized Map implementation for enum keys. It inherits Enum and AbstractMap classes.

```
import java.util.*;
public class EnumMapExample {
  // create an enum
```

```
    public enum Days {
    Monday, Tuesday, Wednesday, Thursday
    };
    public static void main(String[] args) {
    //create and populate enum map
    EnumMap<Days, String> map = new EnumMap<Days, String>(Days.class);
    map.put(Days.Monday, "1");
    map.put(Days.Tuesday, "2");
    map.put(Days.Wednesday, "3");
    map.put(Days.Thursday, "4");
    // print the map
    for(Map.Entry m:map.entrySet()){
       System.out.println(m.getKey()+" "+m.getValue());
      }
    }
}
```



```
Compiled from "EnumMap.java"
public class java.util.EnumMap<K extends java.lang.Enum<K>, V> extends java.util.AbstractMap<K, V
> implements java.io.Serializable, java.lang.Cloneable {
  public java.util.EnumMap(java.lang.Class<K>);
  public java.util.EnumMap(java.util.EnumMap<K, ? extends V>);
  public java.util.EnumMap(java.util.Map<K, ? extends V>);
  public int size();
  public boolean containsValue(java.lang.Object);
  public boolean containsKey(java.lang.Object);
  public V get(java.lang.Object);
  public V put(K, V);
  public V remove(java.lang.Object);
  public void putAll(java.util.Map<? extends K, ? extends V>);
  public void clear();
  public java.util.Set<K> keySet();
  public java.util.Collection<V> values();
  public java.util.Set<java.util.Map$Entry<K, V>> entrySet();
  public boolean equals(java.lang.Object);
  public int hashCode();
  public java.util.EnumMap<K, V> clone();
  public java.lang.Object clone() throws java.lang.CloneNotSupportedException;
  public java.lang.Object put(java.lang.Object, java.lang.Object);
  static {};
}
PS C:\Users\hp> _
```

```
PS C:\Users\hp> javap java.lang.Enum
Compiled from "Enum.java"
public abstract class java.lang.Enum<E extends java.lang.Enum<E>> implements java.lang.constant.C
onstable, java.lang.Comparable<E>, java.io.Serializable {
  public final java.lang.String name();
  public final int ordinal();
  protected java.lang.Enum(java.lang.String, int);
  public java.lang.String toString();
  public final boolean equals(java.lang.Object);
  public final int hashCode();
  protected final java.lang.Object clone() throws java.lang.CloneNotSupportedException;
  public final int compareTo(E);
  public final java.lang.Class<E> getDeclaringClass();
  public final java.util.Optional<java.lang.Enum$EnumDesc<E>> describeConstable();
  public static <T extends java.lang.Enum<T>> T valueOf(java.lang.Class<T>, java.lang.String);
  protected final void finalize();
  public int compareTo(java.lang.Object);
}
PS C:\Users\hp> _
```

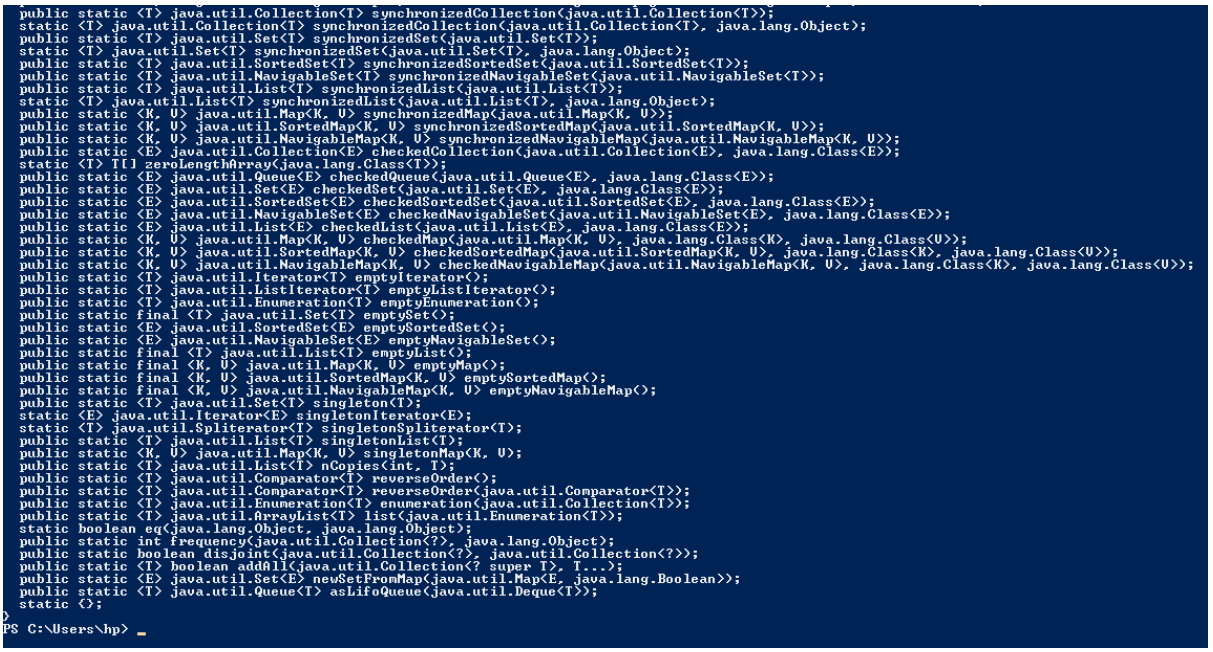
Collections class

Collections class is used exclusively with static methods that operate on or return collections. It inherits the Object class.

The important points about Java Collections class are:

- **Java Collections class supports the polymorphic algorithms that operate on collections.**
- **Java Collections class throws a NullPointerException if the collections or class objects provided to them are null.**

```
PS C:\Users\hp> javap java.util.Collections
Compiled from "Collections.java"
public class java.util.Collections {
  public static final java.util.Set EMPTY_SET;
  public static final java.util.List EMPTY_LIST;
  public static final java.util.Map EMPTY_MAP;
  public static <T extends java.lang.Comparable<? super T>> void sort(java.util.List<T>);
  public static <T> void sort(java.util.List<T>, java.util.Comparator<? super T>);
  public static <T> int binarySearch(java.util.List<? extends java.lang.Comparable<? super T>>, T);
  public static <T> int binarySearch(java.util.List<? extends T>, T, java.util.Comparator<? super T>);
  public static void reverse(java.util.List<?>);
  public static void shuffle(java.util.List<?>);
  public static void shuffle(java.util.List<?>, java.util.Random);
  public static void swap(java.util.List<?>, int, int);
  public static <T> void fill(java.util.List<? super T>, T);
  public static <T> void copy(java.util.List<? super T>, java.util.List<? extends T>);
  public static <T extends java.lang.Comparable<? super T>> T min(java.util.Collection<? extends T>);
  public static <T> T min(java.util.Collection<? extends T>, java.util.Comparator<? super T>);
  public static <T extends java.lang.Comparable<? super T>> T max(java.util.Collection<? extends T>);
  public static <T> T max(java.util.Collection<? extends T>, java.util.Comparator<? super T>);
  public static void rotate(java.util.List<?>, int);
  public static <T> boolean replaceAll(java.util.List<T>, T, T);
  public static int indexOfSubList(java.util.List<?>, java.util.List<?>);
  public static int lastIndexOfSubList(java.util.List<?>, java.util.List<?>);
  public static <T> java.util.Collection<T> unmodifiableCollection(java.util.Collection<? extends T>);
  public static <T> java.util.Set<T> unmodifiableSet(java.util.Set<? extends T>);
  public static <T> java.util.SortedSet<T> unmodifiableSortedSet(java.util.SortedSet<T>);
  public static <T> java.util.NavigableSet<T> unmodifiableNavigableSet(java.util.NavigableSet<T>);
  public static <T> java.util.List<T> unmodifiableList(java.util.List<? extends T>);
  public static <K, V> java.util.Map<K, V> unmodifiableMap(java.util.Map<? extends K, ? extends V>);
  public static <K, V> java.util.SortedMap<K, V> unmodifiableSortedMap(java.util.SortedMap<K, ? extends V>);
  public static <K, V> java.util.NavigableMap<K, V> unmodifiableNavigableMap(java.util.NavigableMap<K, ? extends V>);
  public static <T> java.util.Collection<T> synchronizedCollection(java.util.Collection<T>);
  static <T> java.util.Collection<T> synchronizedCollection(java.util.Collection<T>, java.lang.Object);
  public static <T> java.util.Set<T> synchronizedSet(java.util.Set<T>);
  static <T> java.util.Set<T> synchronizedSet(java.util.Set<T>, java.lang.Object);
  public static <T> java.util.SortedSet<T> synchronizedSortedSet(java.util.SortedSet<T>);
  public static <T> java.util.NavigableSet<T> synchronizedNavigableSet(java.util.NavigableSet<T>);
  public static <T> java.util.List<T> synchronizedList(java.util.List<T>);
  static <T> java.util.List<T> synchronizedList(java.util.List<T>, java.lang.Object);
  public static <K, V> java.util.Map<K, V> synchronizedMap(java.util.Map<K, V>);
  public static <K, V> java.util.SortedMap<K, V> synchronizedSortedMap(java.util.SortedMap<K, V>);
  public static <K, V> java.util.NavigableMap<K, V> synchronizedNavigableMap(java.util.NavigableMap<K, V>);
  public static <E> java.util.Collection<E> checkedCollection(java.util.Collection<E>, java.lang.Class<E>);
  static <T> T[] zeroLengthArray(java.lang.Class<T>);
  public static <E> java.util.Queue<E> checkedQueue(java.util.Queue<E>, java.lang.Class<E>);
  public static <E> java.util.Set<E> checkedSet(java.util.Set<E>, java.lang.Class<E>);
  public static <E> java.util.SortedSet<E> checkedSortedSet(java.util.SortedSet<E>, java.lang.Class<E>);
  public static <E> java.util.NavigableSet<E> checkedNavigableSet(java.util.NavigableSet<E>, java.lang.Class<E>);
  public static <E> java.util.List<E> checkedList(java.util.List<E>, java.lang.Class<E>);
  public static <K, V> java.util.Map<K, V> checkedMap(java.util.Map<K, V>, java.lang.Class<K>, java.lang.Class<V>);
  public static <K, V> java.util.SortedMap<K, V> checkedSortedMap(java.util.SortedMap<K, V>, java.lang.Class<K>, java.lang.Class<V>);
  public static <K, V> java.util.NavigableMap<K, V> checkedNavigableMap(java.util.NavigableMap<K, V>, java.lang.Class<K>, java.lang.Class<V>);
  public static <T> java.util.Iterator<T> emptyIterator();
  public static <T> java.util.ListIterator<T> emptyListIterator();
  public static <T> java.util.Enumeration<T> emptyEnumeration();
  public static final <T> java.util.Set<T> emptySet();
  public static <E> java.util.SortedSet<E> emptySortedSet();
```



Sorting in Collection

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

Collections class provides static methods for sorting the elements of a collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

public void sort(List list): is used to sort the elements of List. List elements must be of the Comparable type.

Note: String class and Wrapper classes implement the Comparable interface. So if you store the objects of string or wrapper classes, it will be Comparable.

Example to sort user-defined class objects

```
import java.util.*;

class Student implements Comparable<Student> {
   public String name;
  public Student(String name) {
    this.name = name;
  }
  public int compareTo(Student person) {
    return name.compareTo(person.name);

  }
}
public class TestSort4 {
  public static void main(String[] args) {
     ArrayList<Student> al=new ArrayList<Student>();
     al.add(new Student("Viru"));
     al.add(new Student("Saurav"));
     al.add(new Student("Mukesh"));
     al.add(new Student("Tahir"));

    Collections.sort(al);
    for (Student s : al) {
     System.out.println(s.name);
    }
  }
}
```

Comparable Interface

- **The Comparable interface is used to order the objects of the user-defined class.**
- **This interface is found in the java.lang package and contains only one method named compareTo(Object).**
- **It provides a single sorting sequence only, i.e., you can sort the elements on the basis of a single data member only. For example, it may be rollno, name, age or anything else.**

compareTo(Object obj) method

public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns

- **positive integer, if the current object is greater than the specified object.**
- **negative integer, if the current object is less than the specified object.**

- **zero, if the current object is equal to the specified object.**

```
class Student implements Comparable<Student>{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}

public int compareTo(Student st){
if(age==st.age)
return 0;
else if(age>st.age)
return 1;
else
return -1;
}
}
```

```
import java.util.*;
public class TestSort1{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

```
PS C:\Users\hp> javap java.lang.Comparable
Compiled from "Comparable.java"
public interface java.lang.Comparable<T> {
  public abstract int compareTo(T);
}
PS C:\Users\hp>
```

Comparator Interface

Comparator interface is used to order the objects of a user-defined class.

This interface is found in the java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Collections class provides static methods for sorting the elements of a collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

public void sort(List list, Comparator c): is used to sort the elements of List by the given Comparator.

Java 8 Comparator interface is a **functional interface** that contains only one abstract method. Now, we can use the Comparator interface as the assignment target for a lambda expression or method reference.

```
PS C:\Users\hp> javap java.util.Comparator
Compiled from "Comparator.java"
public interface java.util.Comparator<T> {
  public abstract int compare(T, T);
  public abstract boolean equals(java.lang.Object);
  public default java.util.Comparator<T> reversed();
  public default java.util.Comparator<T> thenComparing(java.util.Comparator<? super T>);
  public default <U> java.util.Comparator<T> thenComparing(java.util.function.Function<? super T, ? ext
ends U>, java.util.Comparator<? super U>);
  public default <U extends java.lang.Comparable<? super U>> java.util.Comparator<T> thenComparing(java
.util.function.Function<? super T, ? extends U>);
  public default java.util.Comparator<T> thenComparingInt(java.util.function.ToIntFunction<? super T>);

  public default java.util.Comparator<T> thenComparingLong(java.util.function.ToLongFunction<? super T>
);
  public default java.util.Comparator<T> thenComparingDouble(java.util.function.ToDoubleFunction<? supe
r T>);
  public static <T extends java.lang.Comparable<? super T>> java.util.Comparator<T> reverseOrder();
  public static <T extends java.lang.Comparable<? super T>> java.util.Comparator<T> naturalOrder();
  public static <T> java.util.Comparator<T> nullsFirst(java.util.Comparator<? super T>);
  public static <T> java.util.Comparator<T> nullsLast(java.util.Comparator<? super T>);
  public static <T, U> java.util.Comparator<T> comparing(java.util.function.Function<? super T, ? exten
ds U>, java.util.Comparator<? super U>);
  public static <T, U extends java.lang.Comparable<? super U>> java.util.Comparator<T> comparing(java.u
til.function.Function<? super T, ? extends U>);
  public static <T> java.util.Comparator<T> comparingInt(java.util.function.ToIntFunction<? super T>);
  public static <T> java.util.Comparator<T> comparingLong(java.util.function.ToLongFunction<? super T>)
;
  public static <T> java.util.Comparator<T> comparingDouble(java.util.function.ToDoubleFunction<? super
 T>);
}
PS C:\Users\hp> _
```

public int compare(Object obj1, Object obj2) :- It compares the first object with the second object.

public boolean equals(Object obj) :- It is used to compare the current object with the specified object.

Difference between Comparable and Comparator

Comparable and Comparator both are interfaces and can be used to sort collection elements.

However, there are many differences between Comparable and Comparator interfaces that are given below.

Comparable	Comparator

1) Comparable provides a single sorting sequence. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides multiple sorting sequences. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.

2) Comparable affects the original class, i.e., the actual class is modified.	Comparator doesn't affect the original class, i.e., the actual class is not modified.
3) Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.
4) Comparable is present in the java.lang package.	A Comparator is present in the java.util package.
5) We can sort the list elements of Comparable type by Collections.sort(List) method.	We can sort the list elements of Comparator type by Collections.sort(List, Comparator) method.

Properties Class

The properties object contains key and value pairs both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

Advantage of using Properties Class

Recompilation is not required if the information is changed from a properties file: If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

By System.getProperties() method we can get all the properties of the system.

```
PS C:\Users\hp> javap java.util.Properties
Compiled from "Properties.java"
public class java.util.Properties extends java.util.Hashtable<java.lang.Object, java.lang.Object> {
  protected volatile java.util.Properties defaults;
  public java.util.Properties();
  public java.util.Properties(int);
  public java.util.Properties(java.util.Properties);
  public synchronized java.lang.Object setProperty(java.lang.String, java.lang.String);
  public synchronized void load(java.io.Reader) throws java.io.IOException;
  public synchronized void load(java.io.InputStream) throws java.io.IOException;
  public void save(java.io.OutputStream, java.lang.String);
  public void store(java.io.Writer, java.lang.String) throws java.io.IOException;
  public void store(java.io.OutputStream, java.lang.String) throws java.io.IOException;
  public synchronized void loadFromXML(java.io.InputStream) throws java.io.IOException, java.util.InvalidPropertiesFormatException;
  public void storeToXML(java.io.OutputStream, java.lang.String) throws java.io.IOException;
  public void storeToXML(java.io.OutputStream, java.lang.String, java.lang.String) throws java.io.IOException;
  public void storeToXML(java.io.OutputStream, java.lang.String, java.nio.charset.Charset) throws java.io.IOException;
  public java.lang.String getProperty(java.lang.String);
  public java.lang.String getProperty(java.lang.String, java.lang.String);
  public java.util.Enumeration<?> propertyNames();
  public java.util.Set<java.lang.String> stringPropertyNames();
  public void list(java.io.PrintStream);
  public void list(java.io.PrintWriter);
  public int size();
  public boolean isEmpty();
  public java.util.Enumeration<java.lang.Object> keys();
  public java.util.Enumeration<java.lang.Object> elements();
  public boolean contains(java.lang.Object);
  public boolean containsValue(java.lang.Object);
  public boolean containsKey(java.lang.Object);
  public java.lang.Object get(java.lang.Object);
  public synchronized java.lang.Object put(java.lang.Object, java.lang.Object);
  public synchronized java.lang.Object remove(java.lang.Object);
  public synchronized void putAll(java.util.Map<?, ?>);
  public synchronized void clear();
  public synchronized java.lang.String toString();
  public java.util.Set<java.lang.Object> keySet();
  public java.util.Collection<java.lang.Object> values();
  public java.util.Set<java.util.Map$Entry<java.lang.Object, java.lang.Object>> entrySet();
  public synchronized boolean equals(java.lang.Object);
  public synchronized int hashCode();
  public java.lang.Object getOrDefault(java.lang.Object, java.lang.Object);
  public synchronized void forEach(java.util.function.BiConsumer<? super java.lang.Object, ? super java.lang.Object>);
  public synchronized void replaceAll(java.util.function.BiFunction<? super java.lang.Object, ? super java.lang.Object, ?>);
  public synchronized java.lang.Object putIfAbsent(java.lang.Object, java.lang.Object);
  public synchronized boolean remove(java.lang.Object, java.lang.Object);
  public synchronized boolean replace(java.lang.Object, java.lang.Object, java.lang.Object);
  public synchronized java.lang.Object replace(java.lang.Object, java.lang.Object);
  public synchronized java.lang.Object computeIfAbsent(java.lang.Object, java.util.function.Function<? super java.lang.Object, ?>);
  public synchronized java.lang.Object computeIfPresent(java.lang.Object, java.util.function.BiFunction<? super java.lang.Object, ? super java.lang.Object, ?>);
  public synchronized java.lang.Object compute(java.lang.Object, java.util.function.BiFunction<? super java.lang.Object, ? super java.lang.Object, ?>);
  public synchronized java.lang.Object merge(java.lang.Object, java.lang.Object, java.util.function.BiFunction<? super java.lang.Object, ? super java.lang.Object, ?>);
  protected void rehash();
  public synchronized java.lang.Object clone();
  void writeHashtable(java.io.ObjectOutputStream) throws java.io.IOException;
  void readHashtable(java.io.ObjectInputStream) throws java.io.IOException, java.lang.ClassNotFoundException;
  static {};
}
```


ConcurrentHashMap class

public class ConcurrentHashMap<K,V>

extends AbstractMap<K,V>

implements ConcurrentMap<K,V>, Serializable

ConcurrentMap Interface

```
PS C:\Users\hp> javap java.util.concurrent.ConcurrentMap
Compiled from "ConcurrentMap.java"
public interface java.util.concurrent.ConcurrentMap<K, V> extends java.util.Map<K, V> {
  public default V getOrDefault(java.lang.Object, V);
  public default void forEach(java.util.function.BiConsumer<? super K, ? super V>);
  public abstract V putIfAbsent(K, V);
  public abstract boolean remove(java.lang.Object, java.lang.Object);
  public abstract boolean replace(K, V, V);
  public abstract V replace(K, V);
  public default void replaceAll(java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public default V computeIfAbsent(K, java.util.function.Function<? super K, ? extends V>);
  public default V computeIfPresent(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public default V compute(K, java.util.function.BiFunction<? super K, ? super V, ? extends V>);
  public default V merge(K, V, java.util.function.BiFunction<? super V, ? super V, ? extends V>);
}
PS C:\Users\hp> _
```

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as Hashtable and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details..

ConcurrentLinkedQueue Class

ConcurrentLinkedQueue is an unbounded thread-safe queue which arranges the element in FIFO. New elements are added at the tail of this queue and the elements are added from the head of this queue.

ConcurrentLinkedQueue class and its iterator implements all the optional methods of the Queue and Iterator interfaces.

```
PS C:\Users\hp> javap java.util.concurrent.ConcurrentLinkedQueue
Compiled from "ConcurrentLinkedQueue.java"
public class java.util.concurrent.ConcurrentLinkedQueue<E> extends java.util.AbstractQueue<E> implements java.util.Queue<E>, java.io.Serializable {
  volatile transient java.util.concurrent.ConcurrentLinkedQueue$Node<E> head;
  static final java.lang.invoke.VarHandle ITEM;
  static final java.lang.invoke.VarHandle NEXT;
  public java.util.concurrent.ConcurrentLinkedQueue();
  public java.util.concurrent.ConcurrentLinkedQueue(java.util.Collection<? extends E>);
  public boolean add(E);
  final void updateHead(java.util.concurrent.ConcurrentLinkedQueue$Node<E>, java.util.concurrent.ConcurrentLinkedQueue$Node<E>);
  final java.util.concurrent.ConcurrentLinkedQueue$Node<E> succ(java.util.concurrent.ConcurrentLinkedQueue$Node<E>);
  public boolean offer(E);
  public E poll();
  public E peek();
  java.util.concurrent.ConcurrentLinkedQueue$Node<E> first();
  public boolean isEmpty();
  public int size();
  public boolean contains(java.lang.Object);
  public boolean remove(java.lang.Object);
  public boolean addAll(java.util.Collection<? extends E>);
  public java.lang.String toString();
  public java.lang.Object[] toArray();
  public <T> T[] toArray(T[]);
  public java.util.Iterator<E> iterator();
  public java.util.Spliterator<E> spliterator();
  public boolean removeIf(java.util.function.Predicate<? super E>);
  public boolean removeAll(java.util.Collection<?>);
  public boolean retainAll(java.util.Collection<?>);
  public void clear();
  void forEachFrom(java.util.function.Consumer<? super E>, java.util.concurrent.ConcurrentLinkedQueue$Node<E>);
  public void forEach(java.util.function.Consumer<? super E>);
  static {};
}
PS C:\Users\hp> _
```

11. Date & Time API

The java.time, java.util, java.sql and java.text packages contain classes for representing date and time. Following classes are important for dealing with date in Java.

Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes.

Classical Date Time API Classes

The primary classes before Java 8 release were:

Java.lang.System: The class provides the currentTimeMillis() method that returns the current time in milliseconds. It shows the current date and time in milliseconds from January 1st 1970.

java.util.Date: It is used to show specific instant of time, with a unit of millisecond.

The java.util.Date class represents date and time in java. It provides constructors and methods to deal with date and time in java.

The java.util.Date class implements Serializable, Cloneable and Comparable<Date> interface. It is inherited by java.sql.Date, java.sql.Time and java.sql.Timestamp interfaces.

java.sql.Date: The java.sql.Date class represents the only date in Java. It inherits the java.util.Date class.

The java.sql.Date instance is widely used in the JDBC because it represents the date that can be stored in a database.

java.util.Calendar: It is an abstract class that provides methods for converting between instances and manipulating the calendar fields in different ways.

Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable interface.

java.text.DateFormat: The java.text.DateFormat class provides various methods to format and parse date and time in java in language-independent manner. The

DateFormat class is an abstract class. java.text. The Format is the parent class and java.text.SimpleDateFormat is the subclass of java.text.DateFormat class.

In Java, converting the date into the string is called formatting and vice-versa parsing. In other words, *formatting means date to string,* and *parsing means string to date*.

java.text.SimpleDateFormat: It is a class that is used to format and parse the dates in a predefined manner or user defined pattern.

The java.text.SimpleDateFormat class provides methods to format and parse date and time in java. The SimpleDateFormat is a concrete class for formatting and parsing date which inherits java.text.DateFormat class.

Notice that *formatting means converting date to string* and *parsing means converting string to date*.

SimpleDateFormat(String pattern_args): Instantiates the SimpleDateFormat class using the provided pattern - pattern_args, default date format symbols for the default FORMAT locale.

SimpleDateFormat(String pattern_args, Locale locale_args): Instantiates the SimpleDateFormat class using the provided pattern - pattern_args. For the provided FORMAT Locale, the default date format symbols are - locale_args.

SimpleDateFormat(String pattern_args, DateFormatSymbols formatSymbols): Instantiates the SimpleDateFormat class and uses the provided pattern - pattern_args and the date formatSymbols.

java.util.TimeZone: It represents a time zone offset, and also figures out daylight savings.

Java TimeZone class represents a time zone offset, and also figures out daylight savings. It inherits the Object class.

Drawbacks of existing Date/Time API

1. **Thread safety:** The existing classes such as Date and Calendar do not provide thread safety. Hence it leads to hard-to-debug concurrency issues that are needed to be taken care of by developers. The new Date and Time

APIs of Java 8 provide thread safety and are immutable, hence avoiding the concurrency issue from developers.

2. **Bad API designing:** The classic Date and Calendar APIs do not provide methods to perform basic day-to-day functionalities. The Date and Time classes introduced in Java 8 are ISO-centric and provide a number of different methods for performing operations regarding date, time, duration and periods.
3. **Difficult time zone handling:** To handle the time-zone using classic Date and Calendar classes is difficult because the developers were supposed to write the logic for it. With the new APIs, the time-zone handling can be easily done with Local and ZonedDate/Time APIs.

New Date Time API in Java 8

The new date API helps to overcome the drawbacks mentioned above with the legacy classes. It includes the following classes:

java.time.LocalDate: It represents a year-month-day in the ISO calendar and is useful for representing a date without a time. It can be used to represent a date with only information such as a birth date or wedding date.

LocalDate class is an immutable class that represents Date with a default format of yyyy-mm-dd. It inherits Object class and implements the ChronoLocalDate interface.

java.time.LocalTime: It deals in time only. It is useful for representing human-based time of day, such as movie times, or the opening and closing times of the local library.

Java LocalTime class is an immutable class that represents time with a default format of hour-minute-second. It inherits Object class and implements the Comparable interface.

java.time.LocalDateTime: It handles both date and time, without a time zone. It is a combination of LocalDate with LocalTime.

Java LocalDateTime class is an immutable date-time object that represents a date-time, with the default format as yyyy-MM-dd-HH-mm-ss.zzz. It inherits object class and implements the ChronoLocalDateTime interface.

java.time.MonthDay: MonthDay class is an immutable date-time object that represents the combination of a month and day-of-month. It inherits Object class and implements the Comparable interface.

java.time.ZonedDateTime: It combines the LocalDateTime class with the zone information given in ZoneId class. It represents a complete date time stamp along with timezone information.

ZonedDateTime class is an immutable representation of a date-time with a time-zone. It inherits Object class and implements the ChronoZonedDateTime interface.

ZonedDateTime class is used to store all date and time fields, to a precision of nanoseconds, and a time-zone with a zone offset used to handle ambiguous local date-times.

java.time.OffsetTime: It handles time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

OffsetTime class is an immutable date-time object that represents a time, often viewed as hour-minute-second offset. It inherits Object class and implements the Comparable interface.

java.time.OffsetDateTime: It handles a date and time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

OffsetDateTime class is an immutable representation of a date-time with an offset. It inherits Object class and implements the Comparable interface.

OffsetDateTime class is used to store the date and time fields, to a precision of nanoseconds.

java.time.Clock : It provides access to the current instant, date and time in any given time-zone. Although the use of the Clock class is optional, this feature allows us to test your code for other time zones, or by using a fixed clock, where time does not change.

Clock class is used to provide an access to the current, date and time using a time zone. It inherits the Object class.

java.time.Instant : It represents the start of a nanosecond on the timeline (since EPOCH) and useful for generating a timestamp to represent machine time. An instant that occurs before the epoch has a negative value, and an instant that occurs after the epoch has a positive value.

Instant class is used to represent the specific moment on the timeline. It inherits the Object class and implements the Comparable interface.

java.time.Duration : Difference between two instants and measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days, though the class provides methods that convert to days, hours, and minutes.

Duration class is used to measure time in seconds and nanoseconds. It inherits the Object class and implements the Comparable interface.

java.time.Period : It is used to define the difference between dates in date-based values (years, months, days).

Period class is used to measure time in years, months and days. It inherits the Object class and implements the ChronoPeriod interface.

java.time.ZoneId : It states a time zone identifier and provides rules for converting between an Instant and a LocalDateTime.

ZoneId class specifies a time zone identifier and provides a rule for converting between an Instant and a LocalDateTime. It inherits Object class and implements the Serializable interface.

There are two sorts of ID:

1. **Fixed offsets are fully resolved offsets from UTC/Greenwich that apply to all local date-times.**
2. **Geographical regions are areas where a set of rules for determining the offset from UTC/Greenwich apply.**

java.time.ZoneOffset : It describes a time zone offset from Greenwich/UTC time.

ZoneOffset class is used to represent the fixed zone offset from UTC time zone. It inherits the ZoneId class and implements the Comparable interface. The ZoneOffset class declares three constants:

- **MAX: It is the maximum supported zone offsets.**
- **MIN: It is the minimum supported zone offsets.**
- **UTC: It is the time zone offset constant for UTC.**

java.time.format.DateTimeFormatter : It comes up with various predefined formatters, or we can define our own. It has a parse() or format() method for parsing and formatting the date time values.

java.time.Year : Year class is an immutable date-time object that represents a year. It inherits the Object class and implements the Comparable interface.

java.time.YearMonth : YearMonth class is an immutable date-time object that represents the combination of a year and month. It inherits the Object class and implements the Comparable interface.

DayOfWeek enum

In Java the DayOfWeek is an enum representing the 7 days of the week. In addition with the textual enum name, every day-of-week has an int value.

Month enum

In Java, the Month is an enum representing the 12 months of a year. In addition with the textual enum name, every month-of-year has an integer value.

12. JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.

The java.sql package contains classes and interfaces for JDBC API.

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. **Connect to the database**
2. **Execute queries and update statements to the database**
3. **Retrieve the result received from the database.**

JDBC Driver

JDBC Driver is a software component that enables java applications to interact with the database. There are 4 types of JDBC drivers:

1. **JDBC-ODBC bridge driver**
2. **Native-API driver (partially java driver)**
3. **Network Protocol driver (fully java driver)**
4. **Thin driver (fully java driver)**

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses an ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin drivers.

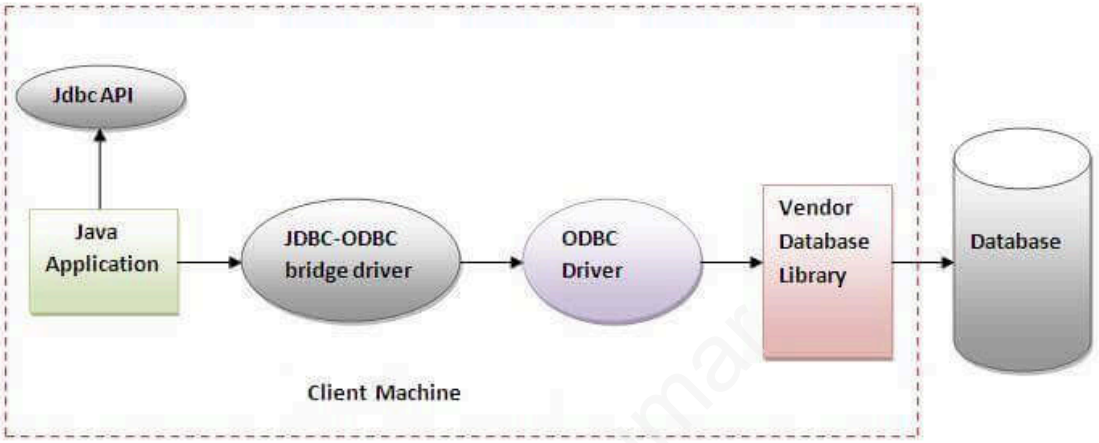


Figure- JDBC-ODBC Bridge Driver

Oracle does not support the JDBC-ODBC Bridge from Java 8.

Advantages:

- **easy to use.**
- **can be easily connected to any database.**

Disadvantages:

- **Performance degraded because JDBC method calls are converted into the ODBC function calls.**
- **The ODBC driver needs to be installed on the client machine.**

2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

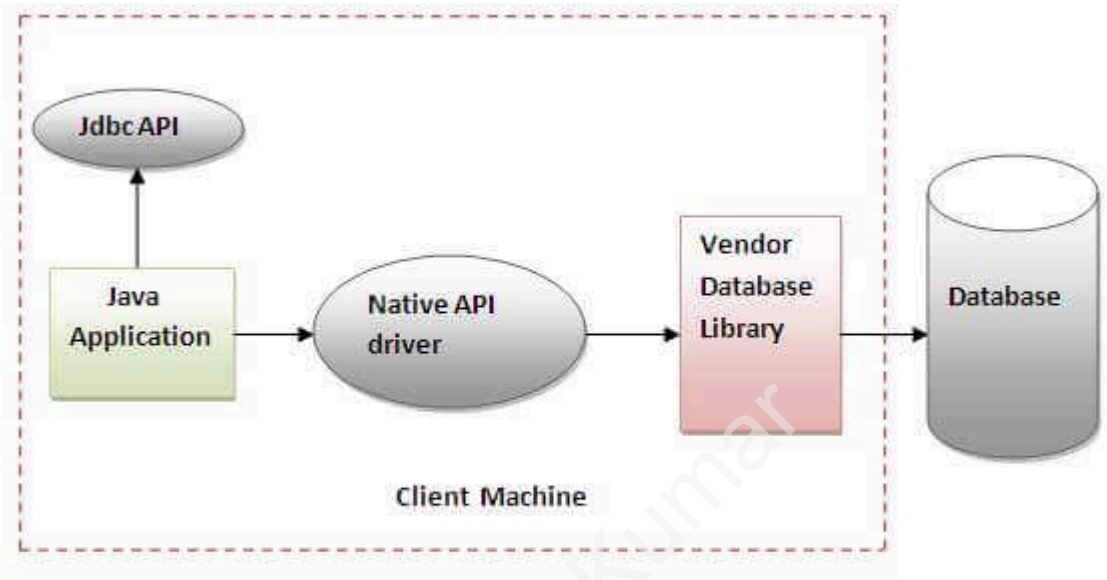


Figure- Native API Driver

Advantage:

- **performance upgraded than JDBC-ODBC bridge driver.**

Disadvantage:

- **The Native driver needs to be installed on each client machine.**
- **The Vendor client library needs to be installed on the client machine.**

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

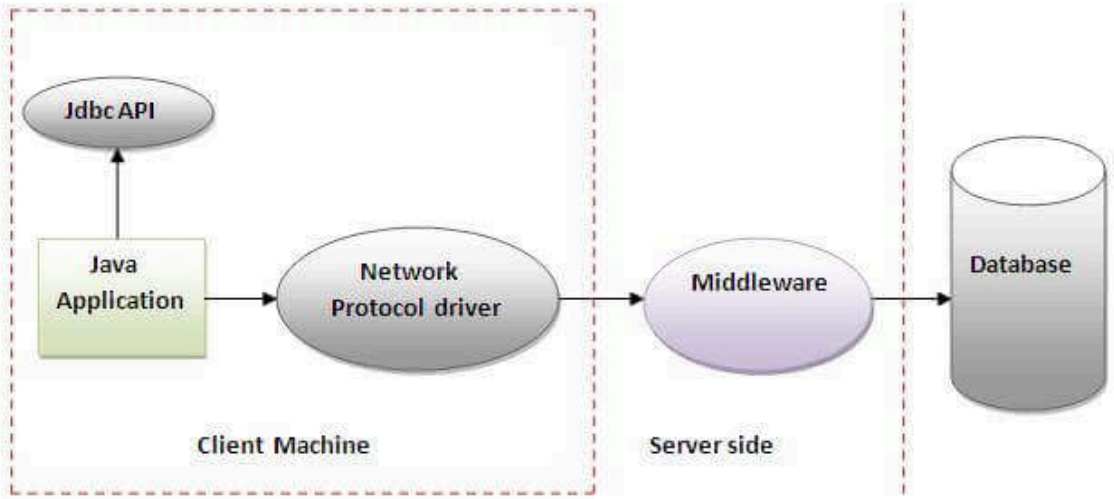


Figure- Network Protocol Driver

Advantage:

- **No client side library is required because of the application server that can perform many tasks like auditing, load balancing, logging etc.**

Disadvantages:

- **Network support is required on client machines.**
- **Requires database-specific coding to be done in the middle tier.**
- **Maintenance of Network Protocol drivers becomes costly because it requires database-specific coding to be done in the middle tier.**

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as a thin driver. It is fully written in Java language.

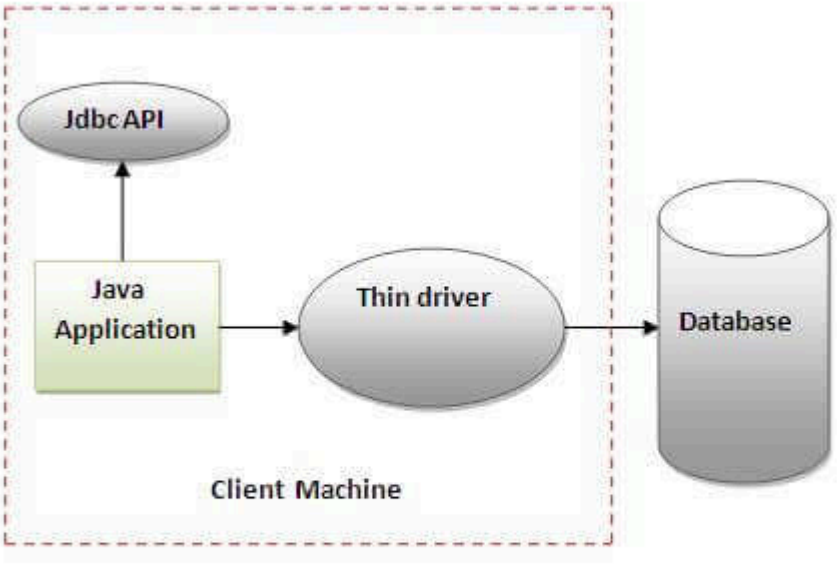


Figure- Thin Driver

Advantage:

- **Better performance than all other drivers.**
- **No software is required at client side or server side.**

Disadvantage:

- **Drivers depend on the Database.**

Database Connectivity with 5 Steps

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- **Register the Driver class**
- **Create connection**
- **Create statement**
- **Execute queries**

- Close connection

1) Register the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

2) Create the connection object

The getConnection() method of the DriverManager class is used to establish connection with the database.

```
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

3) Create the Statement object

The createStatement() method of Connection interface is used to create statements. The object of the statement is responsible to execute queries with the database.

```
Statement stmt=con.createStatement();
```

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

```
ResultSet rs=stmt.executeQuery("select * from emp");

while(rs.next()){
```

```
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

5) Close the connection object

By closing connection object statements and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

```
con.close();
```

```java
package jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class MysqlConnectionExample {

    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            System.out.println("loaded...");
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/vipuldb", "root", "root");
            System.out.println("Connected...");
            Statement stmt = conn.createStatement();
            String sql = "select * from employee";
            ResultSet rs = stmt.executeQuery(sql);
            while(rs.next()) {
                System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " + rs.getInt(3) + " " + rs.getString(4));
            }
            conn.close();
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }

}
```



```
Problems  Javadoc  Declaration  Console  Terminal  Servers
<terminated> MysqlConnectionExample [Java Application] G:\eclipse\plugins\org.eclipse.justj
loaded...
Connected...
1 Vipul Kumar 25 IT
2 Ananya Mohanty 22 Surgery
3 Rohit Kumar 27 IT
```

DriverManager class

The DriverManager class is the component of JDBC API and also a member of the *java.sql* package. The DriverManager class acts as an interface between users and

drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. It contains all the appropriate methods to register and deregister the database driver class and to create a connection between a Java application and the database. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver(). Note that before interacting with a Database, it is a mandatory process to register the driver; otherwise, an exception is thrown.

```
PS C:\Users\hp> javap java.sql.DriverManager
Compiled from "DriverManager.java"
public class java.sql.DriverManager {
  static final java.sql.SQLPermission SET_LOG_PERMISSION;
  static final java.sql.SQLPermission DEREGISTER_DRIVER_PERMISSION;
  public static java.io.PrintWriter getLogWriter();
  public static void setLogWriter(java.io.PrintWriter);
  public static java.sql.Connection getConnection(java.lang.String, java.util.Properties) throws java.sql.SQLException;
  public static java.sql.Connection getConnection(java.lang.String, java.lang.String, java.lang.String) throws java.sql.SQLException;
  public static java.sql.Connection getConnection(java.lang.String) throws java.sql.SQLException;
  public static java.sql.Driver getDriver(java.lang.String) throws java.sql.SQLException;
  public static void registerDriver(java.sql.Driver) throws java.sql.SQLException;
  public static void registerDriver(java.sql.Driver, java.sql.DriverAction) throws java.sql.SQLException;
  public static void deregisterDriver(java.sql.Driver) throws java.sql.SQLException;
  public static java.util.Enumeration<java.sql.Driver> getDrivers();
  public static java.util.stream.Stream<java.sql.Driver> drivers();
  public static void setLoginTimeout(int);
  public static int getLoginTimeout();
  public static void setLogStream(java.io.PrintStream);
  public static java.io.PrintStream getLogStream();
  public static void println(java.lang.String);
  static {};
}
PS C:\Users\hp>
```


Connection interface

A Connection is a session between a Java application and a database. It helps to establish a connection with the database.

The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData, i.e., an object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provides many methods for transaction management like commit(), rollback(), setAutoCommit(), setTransactionIsolation(), etc.

By default, connection commits the changes after executing queries.

```
PS C:\Users\hp> javap java.sql.Connection
Compiled from "Connection.java"
public interface java.sql.Connection extends java.sql.Wrapper,java.lang.AutoCloseable {
  public static final int TRANSACTION_NONE;
  public static final int TRANSACTION_READ_UNCOMMITTED;
  public static final int TRANSACTION_READ_COMMITTED;
  public static final int TRANSACTION_REPEATABLE_READ;
  public static final int TRANSACTION_SERIALIZABLE;
  public abstract java.sql.Statement createStatement() throws java.sql.SQLException;
  public abstract java.sql.PreparedStatement prepareStatement(java.lang.String) throws java.sql.SQLException;
  public abstract java.sql.CallableStatement prepareCall(java.lang.String) throws java.sql.SQLException;
  public abstract java.lang.String nativeSQL(java.lang.String) throws java.sql.SQLException;
  public abstract void setAutoCommit(boolean) throws java.sql.SQLException;
  public abstract boolean getAutoCommit() throws java.sql.SQLException;
  public abstract void commit() throws java.sql.SQLException;
  public abstract void rollback() throws java.sql.SQLException;
  public abstract void close() throws java.sql.SQLException;
  public abstract boolean isClosed() throws java.sql.SQLException;
  public abstract java.sql.DatabaseMetaData getMetaData() throws java.sql.SQLException;
  public abstract void setReadOnly(boolean) throws java.sql.SQLException;
  public abstract boolean isReadOnly() throws java.sql.SQLException;
  public abstract void setCatalog(java.lang.String) throws java.sql.SQLException;
  public abstract java.lang.String getCatalog() throws java.sql.SQLException;
  public abstract void setTransactionIsolation(int) throws java.sql.SQLException;
  public abstract int getTransactionIsolation() throws java.sql.SQLException;
  public abstract java.sql.SQLWarning getWarnings() throws java.sql.SQLException;
  public abstract void clearWarnings() throws java.sql.SQLException;
  public abstract java.sql.Statement createStatement(int, int) throws java.sql.SQLException;
  public abstract java.sql.PreparedStatement prepareStatement(java.lang.String, int, int) throws java.sql.SQLException;
  public abstract java.sql.CallableStatement prepareCall(java.lang.String, int, int) throws java.sql.SQLException;
  public abstract java.util.Map<java.lang.String, java.lang.Class<?>> getTypeMap() throws java.sql.SQLException;
  public abstract void setTypeMap(java.util.Map<java.lang.String, java.lang.Class<?>>) throws java.sql.SQLException;
  public abstract void setHoldability(int) throws java.sql.SQLException;
  public abstract int getHoldability() throws java.sql.SQLException;
  public abstract java.sql.Savepoint setSavepoint() throws java.sql.SQLException;
  public abstract java.sql.Savepoint setSavepoint(java.lang.String) throws java.sql.SQLException;
  public abstract void rollback(java.sql.Savepoint) throws java.sql.SQLException;
  public abstract void releaseSavepoint(java.sql.Savepoint) throws java.sql.SQLException;
  public abstract java.sql.Statement createStatement(int, int, int) throws java.sql.SQLException;
  public abstract java.sql.PreparedStatement prepareStatement(java.lang.String, int, int, int) throws java.sql.SQLException;
  public abstract java.sql.CallableStatement prepareCall(java.lang.String, int, int, int) throws java.sql.SQLException;
  public abstract java.sql.PreparedStatement prepareStatement(java.lang.String, int) throws java.sql.SQLException;
  public abstract java.sql.PreparedStatement prepareStatement(java.lang.String, int[]) throws java.sql.SQLException;
  public abstract java.sql.PreparedStatement prepareStatement(java.lang.String, java.lang.String[]) throws java.sql.SQLException;
  public abstract java.sql.Clob createClob() throws java.sql.SQLException;
  public abstract java.sql.Blob createBlob() throws java.sql.SQLException;
  public abstract java.sql.NClob createNClob() throws java.sql.SQLException;
  public abstract java.sql.SQLXML createSQLXML() throws java.sql.SQLException;
  public abstract boolean isValid(int) throws java.sql.SQLException;
  public abstract void setClientInfo(java.lang.String, java.lang.String) throws java.sql.SQLClientInfoException;
  public abstract void setClientInfo(java.util.Properties) throws java.sql.SQLClientInfoException;
  public abstract java.lang.String getClientInfo(java.lang.String) throws java.sql.SQLException;
  public abstract java.util.Properties getClientInfo() throws java.sql.SQLException;
  public abstract java.sql.Array createArrayOf(java.lang.String, java.lang.Object[]) throws java.sql.SQLException;
  public abstract java.sql.Struct createStruct(java.lang.String, java.lang.Object[]) throws java.sql.SQLException;
  public abstract void setSchema(java.lang.String) throws java.sql.SQLException;
  public abstract java.lang.String getSchema() throws java.sql.SQLException;
  public abstract void abort(java.util.concurrent.Executor) throws java.sql.SQLException;
  public abstract void setClientInfo(java.util.Properties) throws java.sql.SQLClientInfoException;
  public abstract java.lang.String getClientInfo(java.lang.String) throws java.sql.SQLException;
  public abstract java.util.Properties getClientInfo() throws java.sql.SQLException;
  public abstract java.sql.Array createArrayOf(java.lang.String, java.lang.Object[]) throws java.sql.SQLException;
  public abstract java.sql.Struct createStruct(java.lang.String, java.lang.Object[]) throws java.sql.SQLException;
  public abstract void setSchema(java.lang.String) throws java.sql.SQLException;
  public abstract java.lang.String getSchema() throws java.sql.SQLException;
  public abstract void abort(java.util.concurrent.Executor) throws java.sql.SQLException;
  public abstract void setNetworkTimeout(java.util.concurrent.Executor, int) throws java.sql.SQLException;
  public abstract int getNetworkTimeout() throws java.sql.SQLException;
  public default void beginRequest() throws java.sql.SQLException;
  public default void endRequest() throws java.sql.SQLException;
  public default boolean setShardingKeyIfValid(java.sql.ShardingKey, java.sql.ShardingKey, int) throws java.sql.SQLException;
  public default boolean setShardingKeyIfValid(java.sql.ShardingKey, int) throws java.sql.SQLException;
  public default void setShardingKey(java.sql.ShardingKey, java.sql.ShardingKey) throws java.sql.SQLException;
  public default void setShardingKey(java.sql.ShardingKey) throws java.sql.SQLException;
}
PS C:\Users\hp> _
```

Statement interface

The Statement interface provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides a factory method to get the object of ResultSet.

```
PS C:\Users\hp> javap java.sql.Statement
Compiled from "Statement.java"
public interface java.sql.Statement extends java.sql.Wrapper,java.lang.AutoCloseable {
  public static final int CLOSE_CURRENT_RESULT;
  public static final int KEEP_CURRENT_RESULT;
  public static final int CLOSE_ALL_RESULTS;
  public static final int SUCCESS_NO_INFO;
  public static final int EXECUTE_FAILED;
  public static final int RETURN_GENERATED_KEYS;
  public static final int NO_GENERATED_KEYS;
  public abstract java.sql.ResultSet executeQuery(java.lang.String) throws java.sql.SQLException;
  public abstract int executeUpdate(java.lang.String) throws java.sql.SQLException;
  public abstract void close() throws java.sql.SQLException;
  public abstract int getMaxFieldSize() throws java.sql.SQLException;
  public abstract void setMaxFieldSize(int) throws java.sql.SQLException;
  public abstract int getMaxRows() throws java.sql.SQLException;
  public abstract void setMaxRows(int) throws java.sql.SQLException;
  public abstract void setEscapeProcessing(boolean) throws java.sql.SQLException;
  public abstract int getQueryTimeout() throws java.sql.SQLException;
  public abstract void setQueryTimeout(int) throws java.sql.SQLException;
  public abstract void cancel() throws java.sql.SQLException;
  public abstract java.sql.SQLWarning getWarnings() throws java.sql.SQLException;
  public abstract void clearWarnings() throws java.sql.SQLException;
  public abstract void setCursorName(java.lang.String) throws java.sql.SQLException;
  public abstract boolean execute(java.lang.String) throws java.sql.SQLException;
  public abstract java.sql.ResultSet getResultSet() throws java.sql.SQLException;
  public abstract int getUpdateCount() throws java.sql.SQLException;
  public abstract boolean getMoreResults() throws java.sql.SQLException;
  public abstract void setFetchDirection(int) throws java.sql.SQLException;
  public abstract int getFetchDirection() throws java.sql.SQLException;
  public abstract void setFetchSize(int) throws java.sql.SQLException;
  public abstract int getFetchSize() throws java.sql.SQLException;
  public abstract int getResultSetConcurrency() throws java.sql.SQLException;
  public abstract int getResultSetType() throws java.sql.SQLException;
  public abstract void addBatch(java.lang.String) throws java.sql.SQLException;
  public abstract void clearBatch() throws java.sql.SQLException;
  public abstract int[] executeBatch() throws java.sql.SQLException;
  public abstract java.sql.Connection getConnection() throws java.sql.SQLException;
  public abstract boolean getMoreResults(int) throws java.sql.SQLException;
  public abstract java.sql.ResultSet getGeneratedKeys() throws java.sql.SQLException;
  public abstract int executeUpdate(java.lang.String, int) throws java.sql.SQLException;
  public abstract int executeUpdate(java.lang.String, int[]) throws java.sql.SQLException;
  public abstract int executeUpdate(java.lang.String, java.lang.String[]) throws java.sql.SQLException;
  public abstract boolean execute(java.lang.String, int) throws java.sql.SQLException;
  public abstract boolean execute(java.lang.String, int[]) throws java.sql.SQLException;
  public abstract boolean execute(java.lang.String, java.lang.String[]) throws java.sql.SQLException;
  public abstract int getResultSetHoldability() throws java.sql.SQLException;
  public abstract boolean isClosed() throws java.sql.SQLException;
  public abstract void setPoolable(boolean) throws java.sql.SQLException;
  public abstract boolean isPoolable() throws java.sql.SQLException;
  public abstract void closeOnCompletion() throws java.sql.SQLException;
  public abstract boolean isCloseOnCompletion() throws java.sql.SQLException;
  public default long getLargeUpdateCount() throws java.sql.SQLException;
  public default void setLargeMaxRows(long) throws java.sql.SQLException;
  public default long getLargeMaxRows() throws java.sql.SQLException;
  public default long[] executeLargeBatch() throws java.sql.SQLException;
  public default long executeLargeUpdate(java.lang.String) throws java.sql.SQLException;
  public default long executeLargeUpdate(java.lang.String, int) throws java.sql.SQLException;
  public default long executeLargeUpdate(java.lang.String, int[]) throws java.sql.SQLException;
  public default long executeLargeUpdate(java.lang.String, java.lang.String[]) throws java.sql.SQLException;
  public default java.lang.String enquoteLiteral(java.lang.String) throws java.sql.SQLException;
  public default java.lang.String enquoteIdentifier(java.lang.String, boolean) throws java.sql.SQLException;
  public default boolean isSimpleIdentifier(java.lang.String) throws java.sql.SQLException;
  public default java.lang.String enquoteNCharLiteral(java.lang.String) throws java.sql.SQLException;
}
PS C:\Users\hp>
```

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, the cursor points to the first row.

ut we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
```

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in the result set object.
4) public boolean last():	is used to move the cursor to the last row in the result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.

7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of the specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of the specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of the specified column name of the current row as String.

PreparedStatement interface

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized queries.

Let's see the example of parameterized query:

```
String sql="insert into emp values(?,?,?)";
```

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

Improves performance: The performance of the application will be faster if you use the PreparedStatement interface because the query is compiled only once.

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement.

ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of columns, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

The getMetaData() method of the ResultSet interface returns the object of ResultSetMetaData.

DatabaseMetaData interface

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

The getMetaData() method of Connection interface returns the object of DatabaseMetaData.

Store image in database

You can store images in the database in java by the help of **PreparedStatement interface**.

The **setBinaryStream()** method of PreparedStatement is used to set Binary information into the parameterIndex.

For storing images into the database, BLOB (Binary Large Object) & LONGBLOB datatype is used in the table.

Retrieve image from database

By the help of PreparedStatement we can retrieve and store the image in the database.

The getBlob() method of PreparedStatement is used to get Binary information, it returns the instance of Blob. After calling the getBytes() method on the blob object, we can get the array of binary information that can be written into the image file.

```
public Blob getBlob()throws SQLException
```

```
public  byte[] getBytes(long pos, int length)throws SQLException
```

Store file in database

The setCharacterStream() method of PreparedStatement is used to set character information into the parameterIndex.

For storing files into the database, CLOB (Character Large Object) datatype is used in the table.

Retrieve file from database

The getClob() method of PreparedStatement is used to get file information from the database.

CallableStatement Interface

CallableStatement interface is used to call the stored procedures and functions.

We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.

Suppose you need to get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

The prepareCall() method of Connection interface returns the instance of CallableStatement.

```
CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");
```

Example to call the function using JDBC

We use the registerOutParameter method of CallableStatement interface, that registers the output parameter with its corresponding type.

Transaction Management in JDBC

Transaction represents a single unit of work.

The ACID properties describe the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

Atomicity means either all are successful or none.

Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that a transaction is isolated from other transactions.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

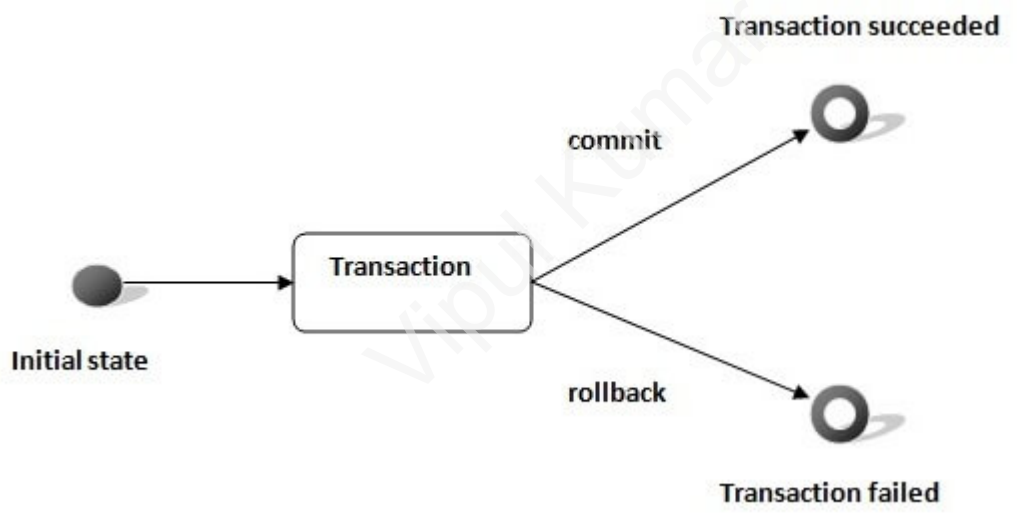
Advantage of Transaction Management

Fast performance: It makes the performance fast because the database is hit at the time of commit.

In JDBC, the Connection interface provides methods to manage transactions.

Method	Description

void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.



Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast. It is because when one sends multiple statements of SQL at once to the database, the communication overhead is reduced significantly, as one is not communicating with the database frequently, which in turn results in fast performance.

The java.sql.Statement and java.sql.PreparedStatement interfaces provide methods for batch processing.

Example of batch processing in JDBC

Let's see a simple example of batch processing in JDBC. It follows following steps:

- **Load the driver class**
- **Create Connection**
- **Create Statement**
- **Add query in the batch**
- **Execute Batch**
- **Close Connection**

JDBC RowSet

An instance of RowSet is the Java bean component because it has properties and Java bean notification mechanism. It is the wrapper of ResultSet. A JDBC RowSet facilitates a mechanism to keep the data in tabular form. It happens to make the data more flexible as well as easier as compared to a ResultSet. The connection between the data source and the RowSet object is maintained throughout its life cycle. The RowSet supports development models that are component-based such as JavaBeans, with the standard set of properties and the mechanism of event notification.

The implementation classes of the RowSet interface are as follows:

- **JdbcRowSet**
- **CachedRowSet**

- WebRowSet
- JoinRowSet
- FilteredRowSet

Let's see how to create and execute RowSet.

```
JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
rowSet.setUsername("system");
rowSet.setPassword("oracle");

rowSet.setCommand("select * from emp400");
rowSet.execute();
```

Advantage of RowSet

The advantages of using RowSet are given below:

1. It is easy and flexible to use.
2. It is Scrollable and Updatable by default.