**CORE JAVA**

## *Introduction:*

**JAVA** was developed by James Gosling at **Sun Microsystems Inc** in the year **1995** and later acquired by Oracle Corporation.

JAVA is a high-level, class-based, **object-oriented programming language** and a **platform**. It is mainly used for application programming i.e. It is widely used in Windows-based (Standalone applications), web-based, enterprise and mobile app development.

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

* Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustments. This is all possible because of JVM. So, its platform Independent

**PLATFORM** ---> A platform is a hardware or software environment in which a program runs. JAVA has a platform i.e; JRE

**PATH vs CLASSPATH:**

Path is set for java tools in java programs like java and javac, which are used to compile your code.

CLASSPATH is used by System or Application class loader to locate and load compile Java bytecodes stored in the .class file.

* The **sourcepath** is the path to the sources you are compiling.
* The **classpath** is a path (or multiple paths) to libraries you are compiling against. These are compiled classes, either in folders or Jar files.
* The **buildpath** is used by the compiler to resolve dependencies and build a project.

**class** Simple{

**public** **static** **void** main(String args[]){

     System.out.println("Hello Java");

    }

}

**System** is the class name, it is declared as final.

**out** is an instance of the System class and is of type PrintStream. Its access specifiers are public and final.

All instances of the PrintStream class have a public method called **println()**, It is responsible for printing the argument and printing a new line.

**Note**: Yes, we can save a Java source file by another name than the class name, if the class is not public.



ASCII values

0 to 9 ----> 48 to 57

A to Z ----> 65 to 90

a to b ----> 97 to 122

--->to execute code;

Compile: javac Simple.java

Run: java Simple //here, we need to give .class name

Packages:

Packages in Java can be defined as the grouping of related types of classes, interfaces, etc providing access to protection and namespace management.

There are various advantages of defining packages in Java.

* Packages avoid name clashes.
* The Package provides easier access control.
* We can also have the hidden classes that are not visible outside and are used by the package.
* It is easier to locate the related classes.

There are two types of packages in Java

* User-defined packages
* Build In packages

--->to execute packages;

Compile: javac -d . Simple.java (-d means destination & .(dot) means current location/within same directory)

Run: java packageName.Simple

## *Compile & Runtime:*

At compile time, the Java file is compiled by Java Compiler (It does not interact with OS) and converts the **Java code(**source code**)** into **bytecode**(machine code)



At runtime, the **bytecode**(executable code) is started running by JVM. ie; The interpreter executes this bytecode at runtime and produces output.

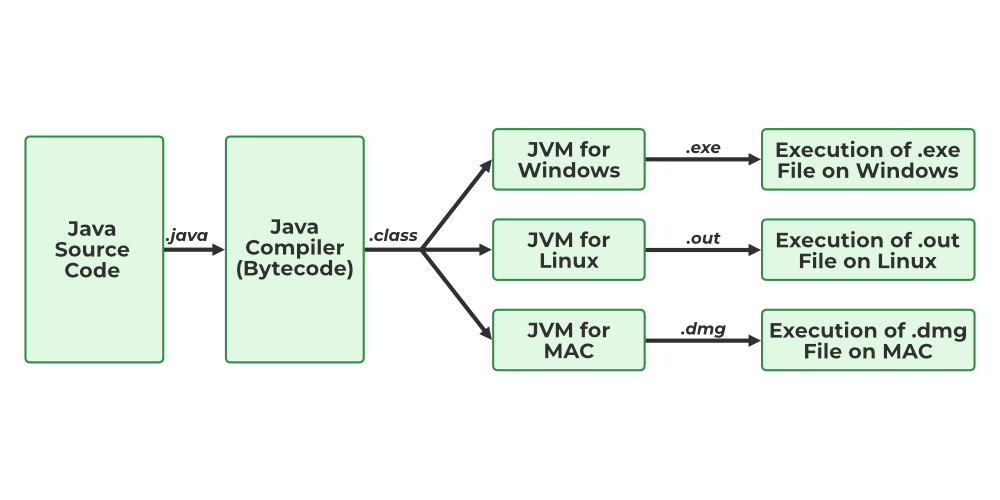
the following steps are performed:



* **Classloader:** It is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the JVM dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
* **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access rights to objects.
* **Interpreter:** Read bytecode stream then execute the instructions.

# *Features of JAVA:*

**1. Platform Independent:**Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, or macOS (means if we compile a program on Windows, then we can run it on Linux ). Any system having JVM can run it irrespective of their operating system. That’s why Java is platform-independent. Byte code is referred to as a Portable code.



**2. Object-Oriented Programming:**Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior. The **four** main concepts are:

* Abstraction
* Encapsulation
* Inheritance
* Polymorphism

**3.** **Simple:**Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, and Explicit memory allocation.

**4.** **Robust:**Java language is robust which means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler can detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

**5.** **Secure:** Java is best known for its security. Witih Java, we can develop virus-free systems. we can directly share an application with the user without sharing the actual program makes Java a secure language. Java is secured because:

- No explicit pointer

- Java Programs run inside a **virtual machine sandbox**



**6.** **Distributed:**We can create distributed applications using the java programming language. Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

**7.** **Multithreading:**Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU.

**8.** **Portable:**Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

**9. High Performance:** Java architecture is defined in such a way that it reduces overhead during the runtime and at sometimes java uses Just In Time (JIT) compiler where the compiler compiles code on-demand basics where it only compiles those methods that are called making applications to execute faster.

Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc but faster than other traditional interpreted programming languages.

**10. Dynamic flexibility:**Java being completely object-oriented gives us the flexibility to add classes, new methods to existing classes, and even create new classes through sub-classes. Java even supports functions written in other languages such as C, C++ which are referred to as native methods.

# *CONSTRUCTOR:*

It is a special type of method which is used to initialize the object. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

There are two rules defined for the constructor;

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

There are **two types of constructors** in Java:

Default constructor (no-arg constructor)

Parameterized constructor

**Constructor Overloading in Java**

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading 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.

# *Keywords:*

Java keywords are known as **reserved words** (predefined words), which hold special meaning. so they cannot be used for variable or object or class names.

* In programming languages, **identifiers** are used for identification purposes. In Java, an identifier can be a class name, method name, variable name, or label generated by programmer.
* **Reserved Word** can’t be used as an identifier.

STATIC:

It is mainly used for memory management(memory efficient/ saves memory).

**Static Variable:**

* static variable can be used to refer to the common property of all objects.
* static variable gets memory only once in the **class area** (its a separate location like heap, stack
* It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class.

**Static Method:**

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

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

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

FINAL KEYWORD --> It is used to **restrict** the user, if you make any variable as final, you cannot change the value of final variable (It will be constant).

final methods cannot be overridden.

Final class cannot be inherited.

//uninitialized final variable can be initialized in the constructor only.

this keyword:

this can be used to refer current class instance variable.

this can be used to invoke current class method (implicitly)

this() can be used to invoke current class constructor.

**NOTE**: return this; //if used inside method, will return current obj

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

***USES:***

super can be used to refer immediate parent class instance variable.

super can be used to invoke immediate parent class method.

super() can be used to invoke immediate parent class constructor.

**Note**: both this() & super() can’t be used together because, those keywords have to be first in the constructor. Since both keywords can't be first at the same time, you can't use them together.

# *C++ vs Java:*

|  |  |  |
| --- | --- | --- |
| **Comparison** | **C++** | **Java** |
| **Platform-independent** | C++ is platform-dependent. | Java is platform-independent. |
| **Mainly used for** | C++ is mainly used for **system** programming. | Java is mainly used for **application** programming. |
| **Libraries** | Comparatively available with low-level functionalities | Wide range of classes for various high-level services |
| **Multiple inheritance** | C++ **supports multiple inheritance.** | Java **doesn't support** multiple inheritance through class. It can be achieved by using [interfaces in java](https://www.javatpoint.com/interface-in-java). |
| **Operator Overloading** | C++ supports [operator overloading](https://www.javatpoint.com/cpp-overloading). | Java doesn't support operator overloading. |
| **Pointers** | C++ supports [pointers](https://www.javatpoint.com/cpp-pointers). You can write a pointer program in C++. | Java supports pointer internally. However, you can't write the pointer program, It 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. |
| **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. |
| **Thread Support** | C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support. | Java has built-in thread support. |
| **Documentation comment** | C++ doesn't support documentation comments. | Java supports documentation comment (/\*\* ... \*/) to create documentation for java source code. |

# *Diff between JDK, JRE, JVM:*



JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other.

1. **Java Development Kit (JDK**) stands for Java Development Kit which provides the environment to develop and execute Java programs. JDK is a package that includes **two things** Development Tools (interpreter/loader (Java), a compiler (javac) etc) to provide an environment to develop your Java programs and, JRE to execute Java programs or applications.

It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.

2.**Java Runtime Environment(JRE)** and also be written as **“Java RTE.”** It is an installation package that provides an environment to **only run** the java program **(not develop)** onto your machine.

It consists of the Java Virtual Machine (JVM), core classes, and supporting files.

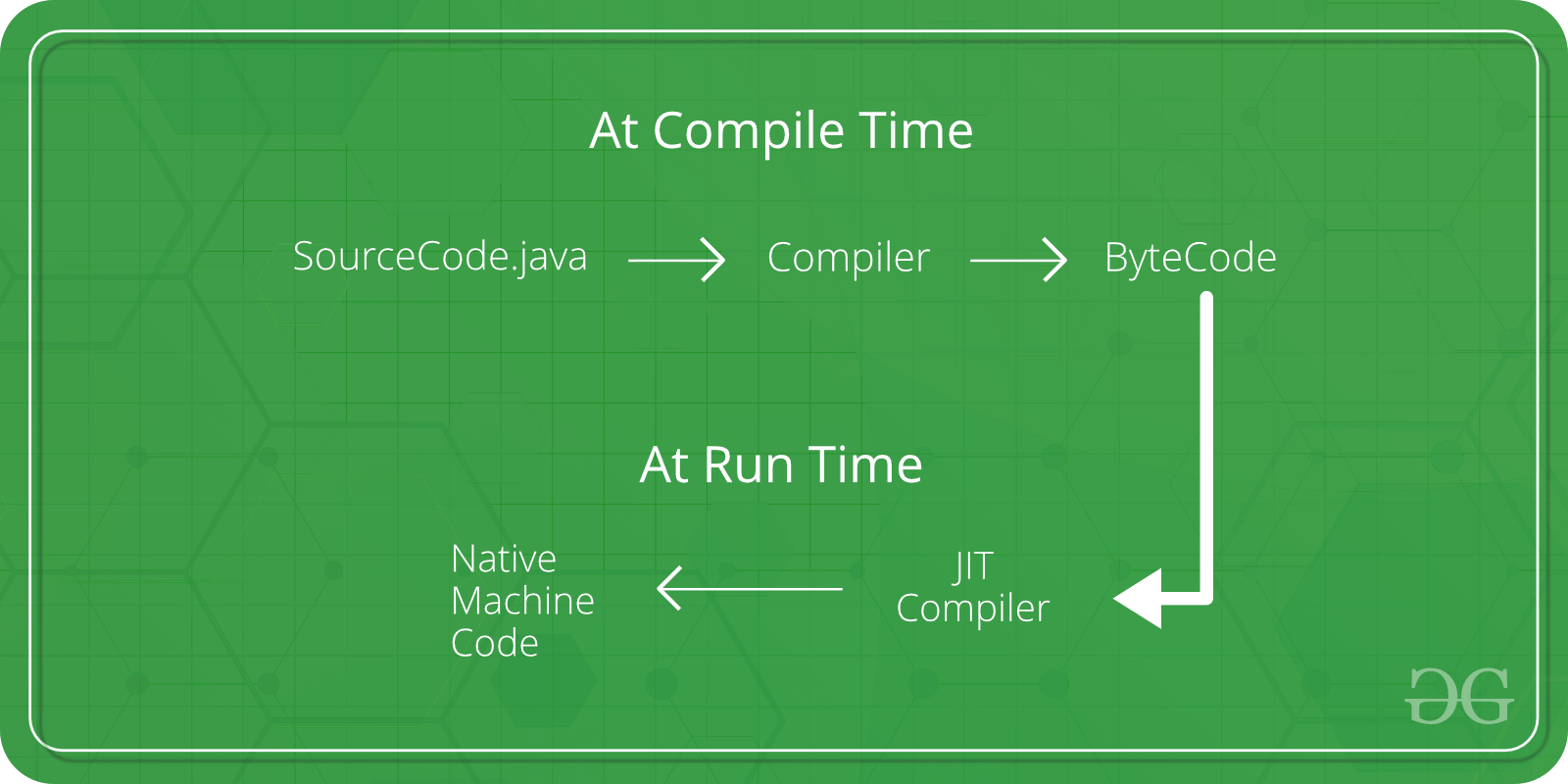
3.[**Java Virtual Machine(JVM)**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) It acts as a run-time engine to run Java applications.  JVM is a type of interpreter responsible for converting bytecode into machine-readable code. JVM is the one that calls the **main** method present in a java code. Whatever Java program you run using JRE or JDK goes into JVM and It is responsible for executing the java program line by line, hence it is also known as an [**i*nterpreter***](https://www.geeksforgeeks.org/compiler-vs-interpreter-2/)**.**

It contains Lang **and util base libraries**, **Integration libraries, User interface toolkits etc.**

**Just-In-Time (JIT) compiler**:

JIT stands for (Just-in-Time) compiler is a part of JRE(Java Runtime Environment), it is used for better performance of the Java applications during **run-time**. To improve performance, JIT compilers interact with the JVM at run time and compile suitable bytecode sequences into native machine code.

* The JIT compiler is enabled throughout, while it gets activated when a method is invoked. For a compiled method, the JVM directly calls the compiled code, instead of interpreting it.
* As JVM calls the compiled code that increases the performance and speed of the execution.



|  |  |
| --- | --- |
| JVM consists of many other components like stack area, heap area, etc. | JIT is **one of the components of JVM** (It present inside JVM). |
| JVM compiles complete byte code to machine code. | JIT compiles only the reusable byte code to machine code. |

# *Access Modifiers:*

These Access modifiers are used to control the scope of classes and methods.

* **Access Modifiers:** default, public, protected, private.
* **Non-access Modifiers:** final, abstract, static, transient, synchronized, volatile, native.

Private: The access level of a private modifier is only **within the class**. It cannot be accessed from outside the class.

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.

Protected: The access level of a protected modifier is **within the package** and **outside the package through child class**.

Public: The access level of a public modifier is **everywhere**.

# *Operations:*

Operators are the special types of symbols used to perform specific operations on OPERANDS.

**Ex:** a+b ---> a=operand, +=operator

There are multiple types of operators in Java all are mentioned below:

1. Arithmetic Operators (+ - \* / %)
2. Unary Operators (- + ! -- ++)
3. Assignment Operator (= += -= \*= /= %=)
4. Relational Operators (== != < <= > >=)
5. Logical Operators (&& || !)
6. Ternary Operator (con ? true : false)
7. Bitwise Operators (& | ~ ^)
8. Shift Operators (<< >> >>>)

LEFT SHIFT ---> System.out.println(10<<2); //10\*2^2=10\*4=40

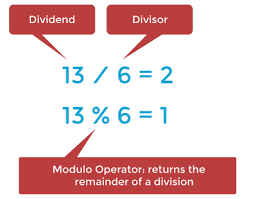
RIGHT SHIFT ---> System.out.println(30>>3); //30/2^3=30/8=3

1. instance of Operator (return Ture /False)

The  **instanceof** operator is used to check whether the object is an instance of the specified type (class or subclass or interface).

 Simple1 s = **new** Simple1();

  System.out.println(s **instanceof** Simple1); //true



the modulo (%) operator gives the **remainder**, while the division (/) operator gives the **quotient**.

Logical && and Bitwise &

The **logical &&** operator doesn't check the 2nd condition if the 1st condition is false. It checks the second condition only if the first one is true.

The **bitwise &** operator always checks both conditions whether 1st condition is true or false.

Logical || and Bitwise |

The **logical ||** operator doesn't check the 2nd condition if the 1st condition is true. It checks the second condition only if the first one is false.

The **bitwise |** operator always checks both conditions whether 1st condition is true or false.

Truth table of the bitwise operators.

| **X** | **Y** | **X & Y     AND** | **X | Y     OR** | **X ^ Y     XOR** |
| --- | --- | --- | --- | --- |
| **0** | **0** | 0 | 0 | 0 |
| **0** | **1** | 0 | 1 | 1 |
| **1** | **0** | 0 | 1 | 1 |
| **1** | **1** | 1 | 1 | 0 |

**Bit Manipulation :**

1. **Get Bit** – fetch the bit present in the given position

Bit Mask: 1 << i // i is the position on bit we want to manipulate

Operation: AND

2. **Set Bit** – set the bit present in the given pos to 1

Bit Mask: 1 << i

Operation: OR

3. **Clear Bit** – set the bit present in the given pos to 0

Bit Mask: 1 << i

Operation: AND with NOT of BitMask

4. **Update Bit**

To update 1 to 0 (same as clear bit)

Bit Mask: 1 << i

Operation: AND with NOT of BitMask

To update 0 to 1 (same as set bit)

Bit Mask: 1 << i

Operation: OR

**--->Java + operator and Operator overloading:**

An operator is said to be overloaded if it can be used to perform more than one functions.

The + operator is overloaded in Java.

However, Java does not support user-defined operator overloading.

The + operator can be used to as an arithmetic addition operator to add numbers and can also be used to concatenate strings.

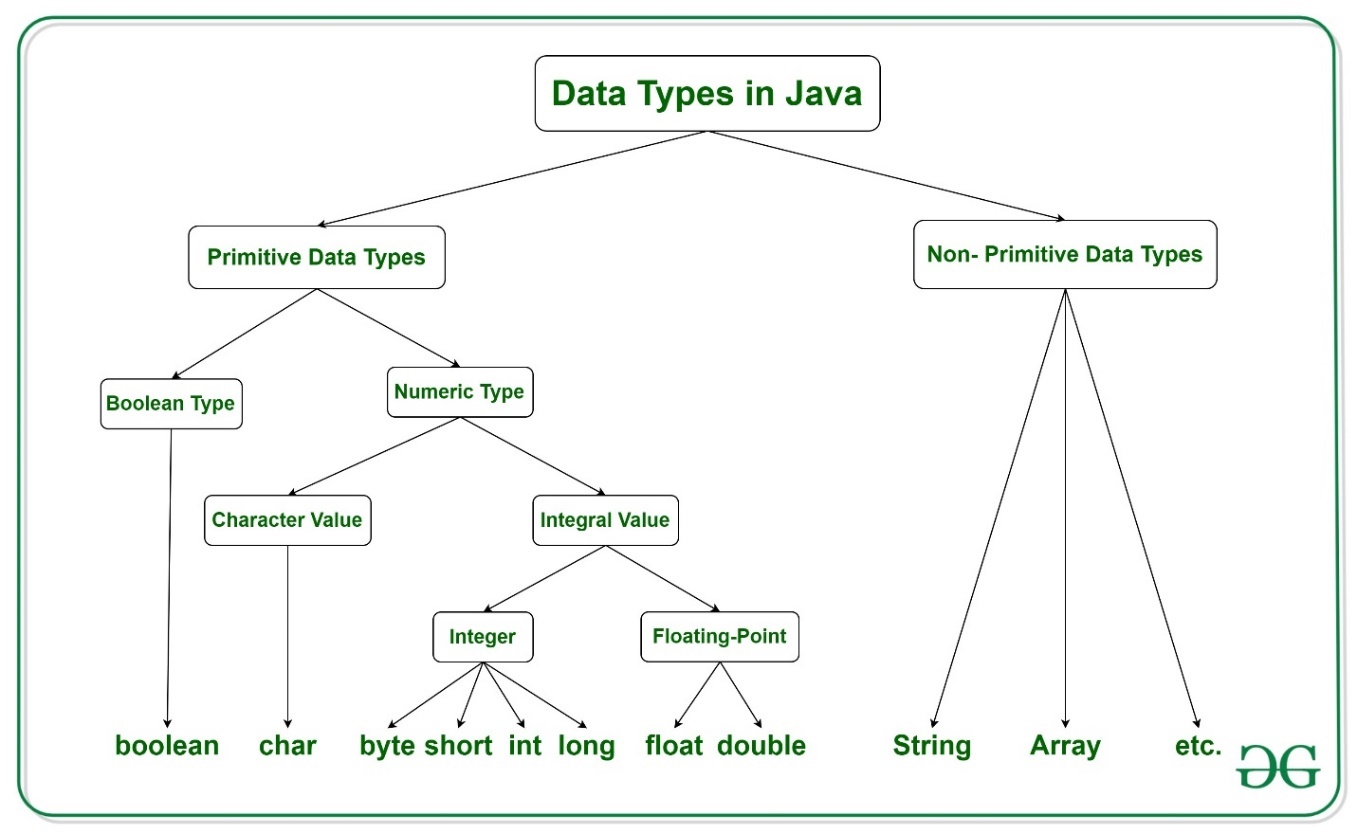
# *Datatypes:*

Datatypes specify the type of values that can be stored in the variable.

1. **Primitive Data Type:** Primitive data are only single values and store only numeric data.   Primitives are stored directly in the memory location. There are 8 primitive data types such as boolean, char, int, short, byte, long, float and double.
2. **Non-Primitive Data Type or Reference Data Types:**

The**se Data Types**will contain a memory address of variable values because the reference types won’t store the variable value directly in memory such as String, Array, objects etc.

// nonprimitives are stored indirectly by referencing a memory location



|  |  |  |
| --- | --- | --- |
| **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 |

**NOTE**: When the reference variable is stored, the **variable** will be stored in the stack and the **original object** will be stored in the heap. In Object data type although two copies will be created, they both will point to the same variable in the heap, hence changes made to any variable will reflect the change in both the variables.

# *Variables:*

VARIABLE is a name of memory location which holds a value. It is the basic unit of storage in a program.

* The value stored in a variable can be changed during program execution.
* Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location.

There are three types of variables in java:

1. **Local Variables**

A variable that is defined within a block or method or constructor is called a local variable.

* These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
* The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
* Initialization of the local variable is mandatory before using it in the defined scope.

2. **Instance Variables**

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

* As instance variables are declared in a class, these variables are created **when an object of the class is created** and destroyed when the object is destroyed.
* Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
* Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null,*for*float*itis*0.0f,*for*int*it is*0,*for Wrapper classes like *Integer* it is *null, etc.*
* Instance variables can be accessed only by creating objects.
* We initialize instance variables using constructors while creating an object.

3. **Static Variables**

Static variables are also known as class variables.

* These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.
* Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
* Static variables are created at the start of program execution and destroyed automatically when execution ends.
* Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null*, for *float* it is *0.0f*, for *int* it is *0*, for *Wrapper classes* like *Integer* it is *null,* etc.
* we access a static variable without need of instance of class.
* Static variables cannot be declared locally inside an instance method.
* Static blocks can be used to initialize static variables.

**Time Complexity:** O(1)  
**Auxiliary Space:** O(1)

**HEAP** 🡪 It is the runtime data area in which objects are allocated. heap saves all objects, arrays and instance variables. This memory is shared between several threads.

**STACK** 🡪 Java Stack stores as frames. It holds local variables (reference variables) and partial results and plays a part in method invocation and return.

# *Wrapper classes:*

The wrapper class is an object class that encapsulates the primitive data types.

wrapper class in Java provides the mechanism to convert primitive data type into object (**Autoboxing**) and object into primitive data type (**Unboxing**).

**Advantages of Wrapper Classes**

1. Collections allowed only object data.
2. On object data we can call multiple methods compareTo(), equals(), toString()
3. Cloning process on objects
4. Object data allows null values.
5. Serialization can allow only object data.

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) |
| int | Integer |
| char | [Character](https://www.javatpoint.com/post/java-character) |
| byte | [Byte](https://www.javatpoint.com/java-byte) |

# *Flow Controls:*

**1. Decision Making statements:**

* if / if-else

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

**2. Loop statements:**

* do while -> checks the condition at the end of the loop after executing the loop statements. use this when the number of iterations are not known and we have to execute the loop at least once
* while -> used to iterate over the number of statements multiple times. use if we don't know the number of iterations in advance.
* for loop -> It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.
* for-each loop -> 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.

**3. Jump statements:**

* break statement -> to come out of a loop. In case of inner loop, it breaks only inner loop.
* continue statement -> It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

# *OOPS concept:*

Oops concept is to improve code readability and reusability by defining a Java program efficiently. The main principles of object-oriented programming are abstraction, encapsulation, inheritance, and polymorphism.

1. **Object**

An Object can be defined as an instance of a class. It is an entity that has state(data) and behavior(functionality). It can be physical or logical. An object contains an address and takes up some space in memory.

1. **Class**

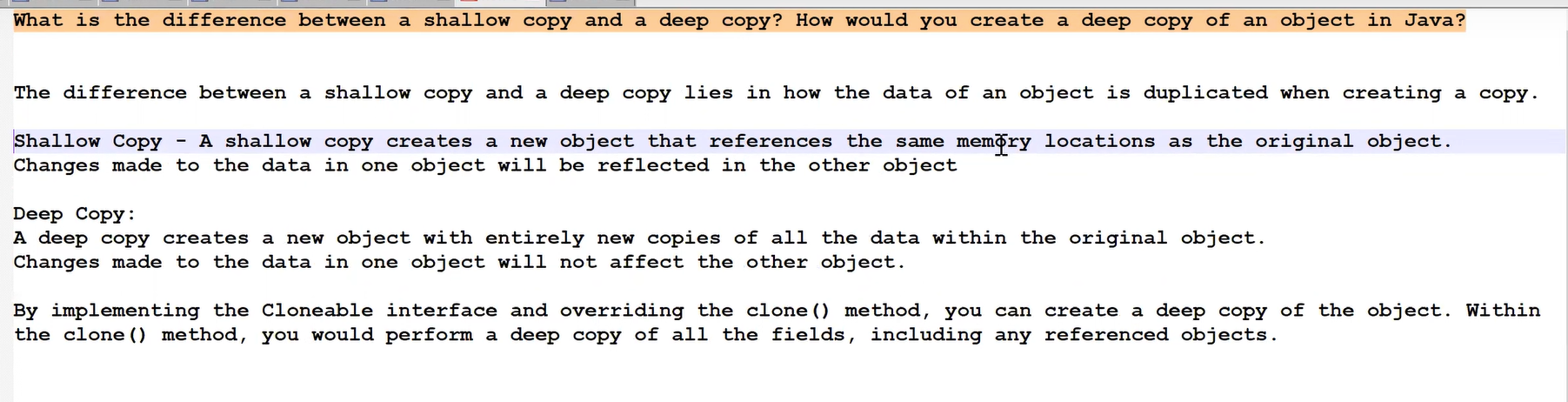
A class is a template/blueprint from which objects are created. It is a logical entity. Class doesn't consume any space.

// A class is a group of objects which have common properties.

The **Object class** is the parent class of all the classes in java by default.

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object. The **java.lang.Cloneable interface** must be implemented by the class, whose object we want to clone.

If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException.**

****

1. **Inheritance**

It is a mechanism in which one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability & method overriding. The **extends keyword** indicates that you are making a new class that derives from an existing class.

**5 TYPES: (IS-A relationship)**

* When a class inherits another class, it is known as a single inheritance
* When there is a chain of inheritance, it is known as multilevel inheritance
* When two or more classes inherits a single class, it is known as hierarchical inheritance
* When a class inherits two classes, it is known as *multiple inheritance*
* Combination of hierarchical and multiple inheritance, is known as *hybrid inheritance*

**AGGREGATION:**

If a class have an entity reference, it is known as Aggregation. It represents **HAS-A relationship.**

Consider a situation, Employee object contains many information’s such as id, name, emailId etc. It contains one more object named **address**, which contains its own information such as city, state, country, zip code etc. In such case, Employee has an entity reference **address**, so relationship is Employee HAS-A address.





* + multiple and hybrid inheritance is supported through interface only.

**Why multiple inheritance is not supported in java?**

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

Since compile-time errors are better than runtime errors, Java renders **compile-time error** if you inherit 2 classes.

The real life example of inheritance is child and parents, all the properties of parents are inherited by his son(child).

1. **Polymorphism**

It is a concept by which we can perform a single action in different ways.

There are two types of polymorphism in Java:

* compile-time polymorphism
* runtime polymorphism.
* If a class has multiple methods having same name but different in parameters, it is known as Method Overloading (COMPILE TIME POLYMORPHISM)

**Can we overload java main() method?**

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

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

Output: main with String[]

There are two ways to overload the method in java:

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

**NOTE**: In Java, Method Overloading is not possible by changing the return type of the method only, we need to change parameters as well. But possible in Method Overriding (covariant return type).

* If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding (RUN TIME POLYMORPHISM)

// we canNOT override java main() method, becoz its a static method.

**Upcasting:** If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. Ex; A a=**new** B();//upcasting

**Note**: Downcasting not possible, it will not give compilation error, but throws **ClassCastException.**

* When **type of the object** is determined at **compiled time** (by the compiler), it is known as static binding.  Dog d1=**new** Dog();
* When **type of the object** is determined at **run-time**, it is known as dynamic binding.  Animal a=**new** Dog();

**Covariant return type:** The covariant return type specifies that the return type may vary in the same direction as the subclass.

since Java5, it is possible to override method by changing the return type if subclass **overrides** any method whose return type is **non-Primitive** but it changes its return type to subclass type

class A{

A get(){return this;}

}

class B extends A{

@Override

B get(){return this;}

void message(){System.out.println("welcome to covariant return type");}

public static void main(String args[]){

new B().get().message();

}

}

Real life example of polymorphism: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So, the same person possess different behavior in different situations

//another ex: Camera, we can take pics, record videos, slow motions, paranoma etc.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No.** | | | **Method Overloading** | **Method Overriding** | |
| 1) | Method overloading is used *to increase the readability* of the program. | | | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. | |
| 2) | Method overloading is performed ***within class***. | | | Method overriding occurs *in* ***two classes*** that have IS-A (inheritance) relationship. | |
| 3) | In case of method overloading, parameter *must be different*. | | | In case of method overriding, *parameter must be same*. | |
| 4) | Method overloading is the example of *compile time polymorphism*. | | | Method overriding is the example of *run time polymorphism*. | |
| 5) | method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading, but you must have to change the parameter. | | | *Return type must be same or covariant* in method overriding. | |

1. **Abstraction**

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

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**. It can have abstract and non-abstract methods. It cannot be instantiated. It can have constructors and static methods also.

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

abstract class Bike{

abstract void run();

}

class Honda extends Bike{

void run(){System.out.println("running safely");}

public static void main(String args[]){

Bike obj = new Honda();

obj.run();

}

}

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

Real life example : ATM machine for cash withdrawal, money transfer, retrieve min-statement, etc in our daily life. But we don't know internally what things are happening inside ATM machine when you insert an ATM card.

**Interface**

Interface is a mechanism to achieve abstraction (100%). There can be **only** abstract methods in the Java interface. It is used to achieve abstraction and **multiple inheritance** in Java (IS-A relationship)

// it can have default and static methods...Static & final variables also

The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.



interface printable{

void print();

}

class A implements printable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

A obj = new A();

obj.print();

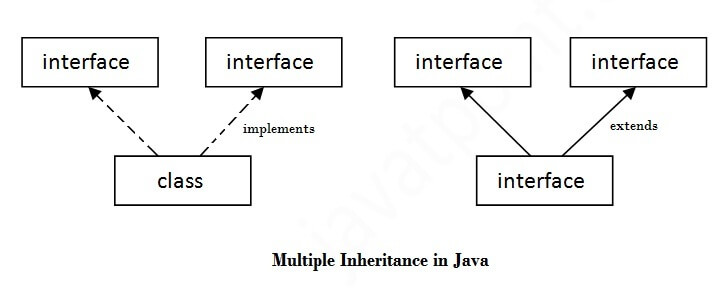
}

}

**Multiple inheritance in Java by interface:**

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

// multiple inheritance can be achieved in Interface because, implementation is done in subclass.



interface Printable{

void print();

}

interface Showable{

void print();

}

class TestInterface3 implements Printable, Showable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

TestInterface3 obj = new TestInterface3();

obj.print();

}

}

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can have **abstract and non-abstract** methods. | Interface can have **only abstract**  methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple interfaces. | An **interface** can extend/implements another interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:**  public abstract class Shape{  abstract void draw();  } | **Example:**  public interface Drawable{  void draw();  } |

**NOTE**: Since Java 8, we can have method body in interface. But we need to make it **default** method.

A Marker interface can be defined as empty interface, having **no data member and functions**, It is called the Marker or **tagged** interface. Ex, Serializable, Cloneable etc

functional interface ----> A functional interface has **only one abstract method,** but it can have multiple default & static methods

**@FunctionalInterface** to make interface as Functional interface. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.

A functional interface can have methods of object class like hascode(), toString() etc.

It can extend another interface only when it doesn’t have any abstract method.

**EX:** Runnable, ActionListener, Comparable etc.

//Lambda exp is used to represent the functional interface

* The Consumer Interface accepts a single argument and does not return any result. It is a functional interface defined in java.util.function package. It contains an abstract accept() and a default andThen() method. It can be used as the assignment target for a lambda expression or method reference.

**6) Encapsulation**

**It** is *a*process of wrapping code(methods) and data(fields) together into a single unit(class)*,* We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

Real life example: Bluetooth connection, we can send data, but we cannot access other persons data

🡪 Suppose you go to an automatic cola vending machine and request for a cola. The machine processes your request and gives the cola. Here automatic cola vending machine is a class. It contains both data i.e. Cola-tin & operations i.e. service mechanism and they are wrapped/integrated under a single unit Cola Vending Machine. This is called **Encapsulation**.

You need not know how the machine is working. This is called **Abstraction**.

You can interact with cola tin only through service mechanism. You cannot access the details about internal data like how much tins it contains, mechanism etc. This is **Data Hiding**.

You cannot pick the tin directly. You request for cola through proper instructions and request mechanism (i.e. by paying amount and filling request) and get that cola only through specified channel. This is **message passing**.

The working and data is hidden from you. This is possible because that Vending machine is made (or Encapsulated or integrated) so. Thus, we can say **Encapsulation is a way to implement Abstraction.**

# *Arrays:*

**An Array** is a collection of similar type (homogenous) of elements which has contiguous memory location. We can store only a fixed set of elements in a Java array (fixed size)

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



1. **int** a[]={33,3,4,5}; //declaration, instantiation and initialization
2. String [] names = new String[3]; //declaration and instantiation

names[0] = "Abbey"; //initialization

1. **int** arr[][]={{1,2,3},{2,4,5},{4,4,5}};
2. **int**[][] arr=**new** **int**[3][3]; //3 row and 3 column

arr[0][0]=1;

arr[0][1]=2;

1. int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };

Advantages

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

Disadvantages

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

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

//declaring a 2D array with odd columns

int arr[][] = new int[3][];

arr[0] = new int[3];

arr[1] = new int[4];

arr[2] = new int[2];

# *Strings:*

String is an object that represents **sequence of characters**. String objects are stored in a special memory area known as the "string constant pool".

Strings are immutable (cannot be changed), Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

**char**[] ch = {'j','a','v','a','t','p','o','i','n','t'};

String s = **new** String(ch);

is same as -> String s = "javatpoint";

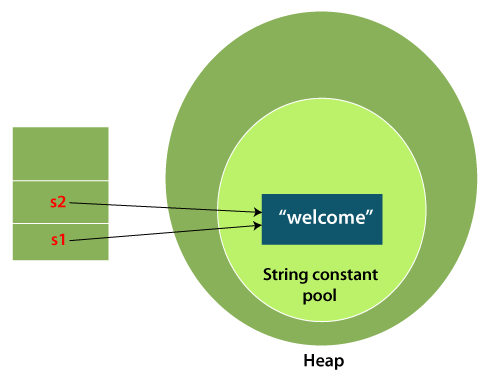
There are two ways to create String object;

**1) By string literal ->** String s="welcome";

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

EX: String s1="Welcome";

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



**# Why Java uses 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");

When you create a string using the **new** keyword, it explicitly creates a new instance of the String class, even if an equivalent string already exists in memory. This means that a new memory allocation is made for the string object, regardless of whether the same string already exists in the string pool.

In such case, JVM will create a new string object in normal heap memory, the variable **s** will refer to the object in a heap (non-pool).

**Immutable String:** Since, strings are immutable, we need to explicitly assign it to the reference variable as shown below.

* class Testimmutablestring{

public static void main(String args[]){

String s="Sachin";

s.concat(" Tendulkar");

System.out.println(s);

}

} o/p: Sachin

* class Testimmutablestring1{

public static void main(String args[]){

String s="Sachin";

s=s.concat(" Tendulkar");

System.out.println(s);

}

} o/p: SachinTendulkar



**Why String objects are immutable in Java?**

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer 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.

String Compare:

There are three ways to compare String in Java:

1. By Using equals() Method
   * equals() method compares the original content of the string. It compares values of string for equality. String class provides **equals(),** **equalsIgnoreCase() methods.**
2. By Using == Operator
   * The == operator compares references not the values.
3. By compareTo() Method
   * compareTo() method compares values lexicographically and returns an integer value if first string is less than, equal to or greater than second string.

**s1 == s2** : returns 0.

**s1 > s2** : returns a positive value //returns the diff in position of alphabets.

**s1 < s2** : returns a negative value //returns the diff in position of alphabets.

String s1="Sachin";

String s2="Sachin";

String s3="Ratan";

System.out.println(s1.compareTo(s2));//0

System.out.println(s1.compareTo(s3));//1(because s1>s3)

System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )

**Concatenation:** It can be achieve by + and concat() method.

String s=50+30+"Sachin"+40+40;

System.out.println(s); //80Sachin4040

**Note: After a string literal, all the + will be treated as string concatenation operator.**

Split()

This method splits the string against given regular expression and returns a char/string array.

--> String[] words=s1.split("\\s+"); //adding + will remove continuous white spaces

**join()** method returns a string combined with a given delimiter.

String s=String.join("-","welcome","to","javatpoint");

System.out.println(s); //welcome-to-javatpoint

**Mutable Strings:**

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings objects.

For these 2 mutable string classes, we use append() method to concatenate the given argument with this String. For regular String we use concat() and +

Ex:

1. StringBuffer sb=**new** StringBuffer("Hello ");
2. sb.append("Java"); //now original string is changed
3. System.out.println(sb); //Hello Java

Available methods are reverse(), insert(int offset, String s), replace(int startIndex, int endIndex, String str), append(String s), capacity() etc.

|  |  |  |
| --- | --- | --- |
| **No.** | **String** | **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 new instance.** | StringBuffer is fast and consumes less memory when we concatenate t strings. |
| 3) | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |
| 4) | String class is slower while performing concatenation operation. | StringBuffer class is faster while performing concatenation operation. |
| 5) | String class uses **String constant pool** | StringBuffer uses **Heap memory** |

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is ***synchronized*** 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. |

**How to create Immutable class?**

We can create immutable class by creating final class that have final data members. All the wrapper classes and String class are immutable.

**Java Regex** ---> It is an API to define a pattern for searching or manipulating strings.

s.o.p(Pattern.matches(".s", "as")); //true

s.o.p(Pattern.matches(".s", "amms")); //false (. means any single char)

s.o.p(Pattern.matches("\\d", "4443")); //false (digit but comes more than once)

s.o.p(Pattern.matches("[a-zA-Z0-9]{6}", "arun32")); //true (count is 6 only)

s.o.p(Pattern.matches("[789]{1}\\d{9}", "8853038949")); //true

//Regular expression to validate a password. A password must start with an alphabet and followed by alphanumeric characters; Its length must be in between 8 to 20. Ans: ^(?=.\*[A-Za-z])(?=.\*\d)[A-Za-z\d]{8,20}$

**Explanation:**

* ^: Start of the string.
* (?=.\*[A-Za-z]): Positive lookahead assertion ensuring that the string contains at least one alphabet (uppercase or lowercase).
* (?=.\*\d): Positive lookahead assertion ensuring that the string contains at least one digit.
* [A-Za-z\d]{8,20}: Match between 8 and 20 characters consisting of alphabets (uppercase or lowercase) and digits.
* $: End of the string.

# *Exception Handling:*

Exception is an abnormal condition (event) that disrupts the normal flow of the program.

The **Exception Handling** is a mechanism to handle the runtime errors, so that the normal flow of the application can be maintained.

Types of Java Exceptions

1. **Checked Exception**

The classes that inherit Throwable class instead RuntimeException, Error are known as checked exceptions. Checked exceptions are checked at compile-time. Ex: IOException, SQLException etc.

1. **Unchecked Exception**

The classes that inherit RuntimeException are known as unchecked exceptions. Unchecked exceptions are not checked at compile-time, but are checked at runtime. Ex: ArithmeticException, NullPointerException, arrayIndexOutOfBoundsException etc

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

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



**NOTE:**

The basic difference between final, finally and finalize is that the ***final*** keyword is an access modifier, ***finally*** is the block in Exception Handling and ***finalize*** is the method of object class (used to perform clean up processing just before object is garbage collected).

**Exception Handling with Method Overriding**

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

**Custom Exception:**

Creating our own Exception is known as custom exception or user-defined exception. In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

class InvalidAgeException extends Exception

{

public InvalidAgeException (String str) {

super(str); // calling the constructor of parent Exception

}

}

public class TestCustomException1

{

static void validate (int age) throws InvalidAgeException{

if(age < 18){

// throw an object of user defined exception

throw new InvalidAgeException("age is not valid to vote");

}

else {

System.out.println("welcome to vote");}

}

public static void main(String args[])

{

try {

validate(13);

}

catch (InvalidAgeException ex) {

System.out.println("Caught the exception");

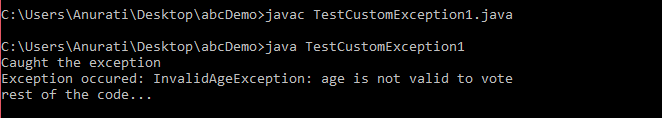
System.out.println("Exception occured: " + ex);

}

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

}

}



# *Multi-threading:*

Multithreading is a process of **executing multiple threads simultaneously**. A thread is a lightweight sub-process. Each thread run in a separate callstack.

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

Advantages:

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

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

3)At a time one thread is executed only. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

4) Multithreading is mostly used in games, animation, etc.

**Life cycle of a Thread (Thread States):**

1) New - Whenever a new thread is created, it is always in the new state

2) 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 (before picked up by thread scheduler)**, and the other is **running (after picked up by thread scheduler)**.

3) Blocked / Waiting - a thread is inactive for a span of time (temporary) if it needs to wait for a resource, such as waiting for I/O operations to complete or waiting to acquire a lock.

4) Timed Waiting - waiting for time leads to starvation, the sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from where it has left earlier.

5) Terminated - A thread reaches the termination state because finishing its job or **Abnormal termination.** You can check if a thread is in the terminated state by calling the isAlive() method.

Thread scheduler in java is the part of the JVM that decides which thread should run. The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

**preemptive scheduling**, the highest priority task executes until it enters the waiting or dead state or even higher priority task comes into existence.

**time slicing**, a task executes for a predefined slice of time and then reenters the pool of ready tasks.

**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.

// If you are not extending the Thread class, your class object would not be treated as a thread object. So, you need to explicitly create Thread class object by passing the object of your class that implements Runnable, so that your class run() method may execute.

* class Multi extends Thread{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

}

}

* class Multi3 implements Runnable{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi3 m1=new Multi3();

Thread t1 =new Thread(m1); // Using the constructor Thread(Runnable r)

t1.start();

}

}

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

* Each thread starts in a separate call stack.
* 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. And it will treated as normal overriden run() method and main method itself will execute the run method , means no multi-threading.

**join():** 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 has achieved its dead state.

* **Daemon thread:**

Daemon threads in Java are special types of threads that **run in the background** and provide services to other threads or perform tasks that don't require explicit termination. **It** is a **service provider thread** that provides services to the user threads. 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 is a low priority thread.

t1.setDaemon(**true**); //now t1 is daemon thread

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

* **Java 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.

* **Thread group**, Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.

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 thread group's parent thread group or any other thread groups.

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");

**Shutdown Hook:**

A shutdown hook is a thread that is registered with the to perform specific actions **when the JVM is shutting down**. These actions could include cleaning up resources, saving data, or performing any necessary cleanup tasks before the program terminates. Common Use Cases:

* Saving unsaved data or state before termination.
* Releasing resources such as closing database connections, file handles, or network sockets.
* Logging shutdown events or performing final cleanup operations.

**Ex:**

public class ShutdownHookExample {

public static void main(String[] args) {

Runtime.getRuntime().addShutdownHook(new Thread(() -> {

// Perform cleanup tasks or save data

System.out.println("Shutdown hook executed");

}));

System.out.println("Application started");

try {

Thread.sleep(5000); // Simulating application running for 5 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

// Terminate the application

System.exit(0);

}

}

**Garbage Collection:**

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory/objects automatically. It makes java **memory efficient** because gc removes the unreferenced objects from heap memory.

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

*Ex: System.gc();*

1) By nulling a reference:

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

2) By assigning a reference to another:

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

3) By anonymous object:

1. **new** Employee();

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

**Synchronization:**

Synchronization has the capability to control the access of multiple threads to any shared resource. Used to prevent thread interference. synchronization is crucial to prevent race conditions, data inconsistencies

**Synchronized Method:**

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

When you declare a method as synchronized, only one thread can execute that method on the object instance at a time. Other threads that attempt to execute the method will be blocked until the synchronized method is released.

class Table{

synchronized void printTable(int n){ //synchronized method

for(int i=1;i<=5;i++){

System.out.println(n\*i);

try{

Thread.sleep(400);

}catch(Exception e){System.out.println(e);}

}

}

}

**Deadlock in Java:**

Deadlock occurs when two or more threads are blocked indefinitely, waiting for each other to release resources that they need to proceed.

In other words, each thread holds a resource that another thread needs, and they are all waiting for each other, resulting in a circular waiting dependency called Deadlock.

Deadlock can happen due to improper resource allocation and synchronization.



public class TestDeadlockExample1 {

public static void main(String[] args) {

final String resource1 = "ratan jaiswal";

final String resource2 = "vimal jaiswal";

**// t1 tries to lock resource1 then resource2**

Thread t1 = new Thread() {

public void run() {

synchronized (resource1) {

System.out.println("Thread 1: locked resource 1");

try { Thread.sleep(100);} catch (Exception e) {}

synchronized (resource2) {

System.out.println("Thread 1: locked resource 2");

}

}

}

};

**// t2 tries to lock resource2 then resource1**

Thread t2 = new Thread() {

public void run() {

synchronized (resource2) {

System.out.println("Thread 2: locked resource 2");

try { Thread.sleep(100);} catch (Exception e) {}

synchronized (resource1) {

System.out.println("Thread 2: locked resource 1");

}

}

}

};

t1.start();

t2.start();

}

}

**Solution for Deadlock:**

The deadlock occurred as the two threads were acquiring the locks in a different order. If we synchronize the thread in the same order, then we can get rid of deadlock.

How to Avoid Deadlock in Java?

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

1. Using Thread. join()
2. Using Synchronization Objects
3. Avoid Nested Locks
4. Avoid Using unnecessary Locks When Not Needed
5. Proper Design of Code

* **Livelock** is similar to deadlock, but instead of threads being blocked indefinitely, they are actively responding to each other, but none of them make progress. In a livelock scenario, two or more threads are stuck in a loop of repeatedly trying to resolve a resource conflict, but their attempts are always unsuccessful. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

**Inter-thread Communication in Java**

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

It is a mechanism in which a synchronized thread is paused running in its critical section and another synchronized 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 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.

**2) notify() method**

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.

**3) notifyAll() method**

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

|  |  |
| --- | --- |
| **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. |

**Serialization:**

Serialization 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 byte-stream is converted into an object (the process of reconstructing the object from the serialized state).

* The serialization and deserialization processes are **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 ObjectOutputStream class, and for deserialization we call the readObject() method of ObjectInputStream class.

Advantages of Java Serialization

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

***NOTE***: ***There are some cases of Serialization with respect to inheritance:***

Case 1: If the superclass is serializable, then subclass is automatically serializable

If the parent class is Serializable, by default all the child classes also Serializable. Hence even though child class doesn’t implement Serializable, we can serialize child class object if parent class implements serializable interface.

Case 2: If the superclass is serializable, but we don’t want the subclass to be serialized

There is no direct way to prevent sub-class from serialization in java. One possible way we can achieve this is by implementing the writeObject() and readObject() methods in the subclass and need to throw NotSerializableException from these methods.

Case 3: If a superclass is not serializable, then subclass can still be serialized

If a superclass is not serializable, to serialize an object of a subclass that inherits from this non-serializable superclass will result in a java.io.NotSerializableException at runtime.

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

* ***Serialization example***

import java.io.Serializable;

public class Student implements Serializable {

int id;

String name;

public Student(int id, String name) {

this.id = id;

this.name = name;

}

}

class Persist{

public static void main(String args[]){

try{

Student s1 =new Student(211,"ravi");

FileOutputStream fout=new FileOutputStream("f.txt");

ObjectOutputStream out=new ObjectOutputStream(fout);

out.writeObject(s1);

out.close(); //closing the stream

System.out.println("success");

}catch(Exception e){System.out.println(e);}

}

}

* ***Deserialization example***

import java.io.\*;

class Depersist{

public static void main(String args[]){

try{

//Creating stream to read the object

ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));

Student s=(Student)in.readObject();

System.out.println(s.id+" "+s.name);

in.close(); //closing the stream

}

catch(Exception e){System.out.println(e);}

}

}

IMP: During the serialization, when we do not want an object to be serialized, we can use a **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 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.

Ex;

**int** id;

 String name;

**transient** **int** age; //Now it will not be serialized

 Student s1 =**new** Student(211,"ravi",22);

After deserialization, o/p will be **211 ravi 0**

# *Collections:*

collections are objects that **group multiple elements** into a single unit. They provide a way to store, retrieve, manipulate, and operate on groups of objects. Java collections framework provides a set of interfaces and classes that offer various implementations of collections, each designed for specific use cases.

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.



**1)List Interface**

List interface is the child 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.

* ArrayList class uses a dynamic array for storing the elements. It is like an array, but there is **no size limit**. We can add or remove elements anytime.

Implements List Interface

can have the duplicate elements

maintains the insertion order

It is non-synchronized i.e, not thread safe

allows random access because the array works on an index basis.

It allows to store the null elements (no 2 nulls are same).

Better for storing and accessing data.

manipulation is little bit slower than the LinkedList because a lot of shifting needs to occur if any element is removed from the array list.

ArrayList<**int**> al = ArrayList<**int**>(); // does not work  for primitive types

ArrayList<Integer> al = **new** ArrayList<Integer>(); // works fine

***NOTE*:** generic collection allows you to have only one type of object in a collection. Now it is type-safe, so typecasting is not required at runtime. In Parameter type we cannot use primitives like ‘int’, ’char’ or ‘double’ it gives compile-time error.

**ArrayList methods:** add(), addAll(), remove(), removeAll(), removeIf(), size(), get(), set(), isEmpty(), clear() etc

**//Traversing list through Iterator**

The Iterator interface provides a way to iterate over a collection in a forward direction only.

Iterator itr=list.iterator();

while(itr.hasNext()){ //check if iterator has the elements

System.out.println(itr.next()); //printing the element and move to next

}

**//Here, element iterates in reverse order**

The ListIterator interface extends Iterator and provides bidirectional traversal over a list. Ie; It allows iterating both forward and backward through a list.

ListIterator<String> list1=list.listIterator(list.size());

while(list1.hasPrevious()) {

System.out.println(list1.previous());

}

**---or----** Iterator i=list.descendingIterator();

while(i.hasNext()) {

System.out.println(i.next());

}

* LinkedList class uses a doubly linked list (add on both sides) to store the elements (A linked list is a linear data structure, in which the elements are not stored at contiguous memory location).

Implements List & Deque Interfaces.

can contain duplicate elements.

maintains insertion order

Allows null elements

It is not synchronized (not thread safe)

manipulation is fast because no shifting needs to occur.

Better for data manipulation



**LinkedList Methods:** add(), addAll(), addFirst(), addLast(), remove(), removeAll(), removeFirst(), removeLast(), peek(), poll(), pop(), clear(), get(), set() etc

|  |  |
| --- | --- |
| **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(**allnodes can be scattered throughout the memory**)** |
| 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. |

* Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is **synchronized** (thread safe) and contains many methods that are not the part of Collection framework. 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, it will perform better in such case.
* The stack is the subclass of Vector. The **stack** is a linear data structure that is used to store the collection of objects based on **Last-In-First-Out** (LIFO) manner. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity. | Vector **increments 100%** ( doubles the array size) if the total number of elements exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2 | Vector is a **legacy** class (It is available since Java 1.0) |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized. |
| 5) ArrayList uses the **Iterator** interface to traverse the elements. | A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements. |

**2)Queue/Deque Interface**

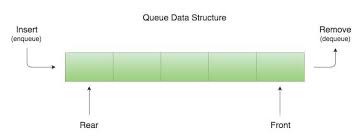
- Priority Queue uses QUEUE and orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.

implements the Queue interface

doesn't maintain the insertion order (sorts based on priority)

Null values are not accepted

Not Syncronized (not thread safe)



**PriorityQueue Methods:** add(), addAll(), offer(), remove(), removeAll(), peek(), poll(), element(), clear() etc

offer() - It is used to insert the specified element into this queue.

Poll() - It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.

Peek() - It is used to retrieves, but does not remove the head of this queue, or returns null if this queue is empty.

Element() - It is used to retrieves, but does not remove the head of this queue.

* Array Deque is a linear collection that uses DEQUE and supports element insertion and removal at both ends Therefore, a deque can be used as a stack or queue. Deque is an acronym for "**double ended queue**"

STACK - LIFO, QUEUE - FIFO

Implements the Deque interface.

Null elements are not allowed

maintains the insertion order

faster than ArrayList, Stack and has no capacity restrictions.

Not Syncronized (not thread safe)

**ArrayDeque Methods:** add(), addAll(), offer(), offerFirst(), offerLast(), remove(), removeAll(), peek(), peekFirst(), peekLast(), poll(), element(), clear() etc

**3)Set Interface**

Set Interface extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. When we create an object of HashSet, it internally creates an instance of HashMap with default initial capacity 16.

* HashSet class implements Set Interface. HashSet class uses a **hash table** for storage.

HashSet stores the elements by using a mechanism called **hashing**.

contains unique elements only.

doesn't maintain the insertion order (Elements are inserted on the basis of their hashcode)

allows atmost one null value.

It is non synchronized (not thread safe)

better for search operations.

The initial default capacity of HashSet is 16, and load factor is 0.75

**HashSet methods:** add(), addAll(), remove(), removeAll(), removeIf(), clear() etc

* LinkedHashSet class extends HashSet and implements the Set interface. Internally, LinkedHashSet uses a hash table and a linked list to store elements.

contains unique elements only.

maintains insertion order

It is non synchronized (not thread safe)

Doesn’t allows null value

Iterating over a LinkedHashSet is generally **slower** than iterating over a HashSet because of the additional linked list maintenance overhead.

**LinkedHashSet methods:** add(), addAll(), remove(), removeAll(), removeIf(), clear() etc

**4)SortedSet Interface**

- TreeSet class implements the Set interface and uses a tree for storage. The objects of the TreeSet class are stored in **ascending order**.

contains unique elements only

doesn’t maintains insertion order //ascending order.

TreeSet class access and retrieval times are quiet fast.

doesn't allow null element.

It is non synchronized (not thread safe)

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 becoz this is there in the **self-balancing tree**. It is there to ensure that the tree height never exceeds O(log(N)) for all 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.

**Treeset Methods:** add(), addAll(), remove(), removeAll(), ceiling(), floor(), higher(), lower(), first(), last(), pollFirst(), polllast(), isEmpty(), size(), clear() etc

Ceiling() - It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.

Floor() - It returns the equal or closest least element of the specified element from the set, or null there is no such element.

Higher() - It returns the closest greatest element of the specified element from the set, or null there is no such element.

Lower() - It returns the closest least element of the specified element from the set, or null there is no such element.

**5)Map Interface**

- HashMap class implements the Map interface which allows us to store elements in key - value pair, where keys should be unique. Each key and value pair is known as an **entry.**  It uses an **Array of LinkedList** data structure internally for storing Key and Value.

It allows one null key and multiple null values.

Hash code of null Key is 0

contains unique keys only (if you try to store duplicate key with another value, it will replace the existing value)

maintains no order (hashcode decides the index location)

Default Hashmap size is 16 with a load factor of 0.75 (When the load factor threshold is reached, the capacity of the HashMap is doubled, and all elements are rehashed and redistributed into the new larger array.)

it is not synchronized (not Thread-safe)

**Hashmap methods:** put(), putAll(), putIfAbsent(), containsKey(), containsValue(), remove(), removeAll(), entrySet(), replace(), size(), isEmpty(), clear() etc

for(Map.Entry m : map.entrySet()){ //Converting to Set so that we can traverse

System.out.println(m.getKey()+" "+m.getValue());

}

map.entrySet()  //Returns a Set view of the mappings

.stream()

.sorted(Map.Entry.comparingByKey()) //sorts based on type

.forEach(System.out::println);

map.entrySet()

.stream()

.sorted(Map.Entry.comparingByValue())

.forEach(System.out::println);

**Internal working of HashMap:**

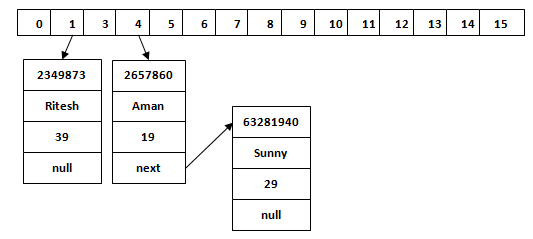
Hashing is the process of converting an **object into an integer** value. The integer value helps in indexing and faster searches.

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, which is the index/ address of the element inside the map.

equals() 🡪  It checks the equality of two objects. It checks that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList.

Buckets 🡪 Array of the node is 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.

Bucket



**Overriding both equals(Object) and hashCode() method:**

You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable.

***General Contracts for hashCode() in Java***

1) If two objects are equal by the equals() method then their hashcode returned by the hashCode() method must be the same.

2) Whenever the hashCode() method is invoked on the same object more than once within a single execution of the application, hashCode() must return the same integer provided no information or fields used in equals and hashcode is modified.

3) If two objects are not equaled by the equals() method it is not required that their hashcode must be different.

**Ex:**

public class Employee {

String name;

int age;

public Employee(String name, int age) {

this.name = name;

this.age = age;

}

//getters and setters

@Override

public boolean equals(Object obj) {

if (obj == this) return true;

if (!(obj instanceof Employee)) return false;

Employee employee = (Employee) obj;

return employee.getAge() == this.getAge()

&& employee.getName() == this.getName();

}

@Override

public int hashCode() {

return Objects.hash(name, age);

}

}

public class ClientTest {

public static void main(String[] args) {

Employee emp = new Employee("rajeev", 24);

Employee emp1 = new Employee("yash", 25);

Employee emp2 = new Employee("rajeev", 24);

HashSet<Employee> employees = new HashSet<Employee>();

employees.add(emp);

employees.add(emp1);

employees.add(emp2);

System.out.println("employee.hashCode(): " + emp.hashCode() + "\n" +

" employee1.hashCode():" + emp1.hashCode() + "\n" +

" employee2.hashCode():" + emp2.hashCode());

//since emp and emp2 are same, equals() override its previous value

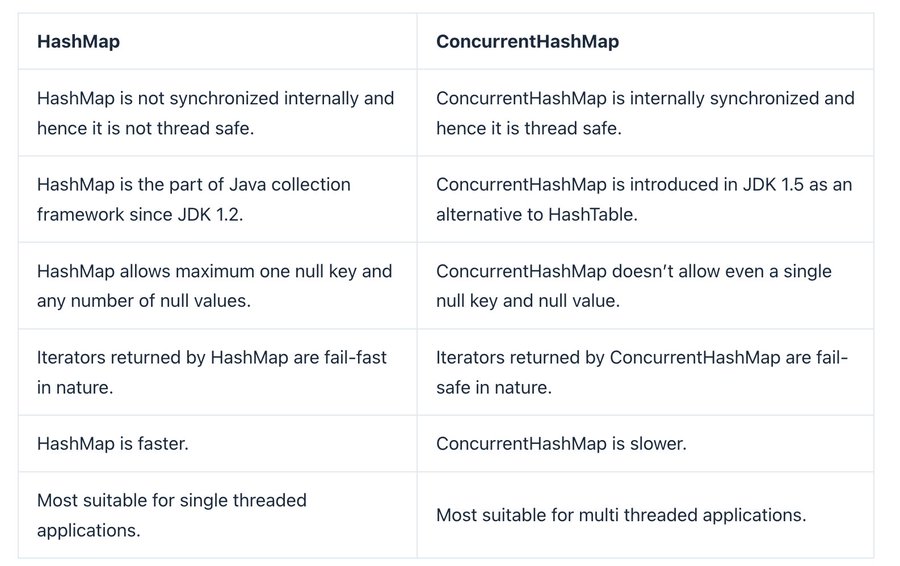
employees.stream().forEach(e ->

System.out.println(e.getName() + "-" + e.getAge()));

}

}

|  |  |
| --- | --- |
| **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**. (java 1.0) |
| 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) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class | Hashtable inherits **Dictionary** class. |



* LinkedHashMap class is Hashtable and Linked list implementation of the Map interface. It inherits HashMap class and implements the Map interface.

contains unique elements.

may have one null key and multiple null values.

maintains insertion order.

It is non synchronized (not thread safe)

initial default capacity of HashMap class is 16, load factor is 0.75

map.keySet() – fetches only keys in map

map.values() – fetches the values

* TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order. It implements the NavigableMap interface and extends AbstractMap class.

contains unique elements.

cannot contain any null key but can have multiple null values.

maintains ascending order.

It is non synchronized (not thread safe)

It is way slower than HashMap because it runs sorting operations with each insertion, update, and removal.

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1) HashMap can contain one null key. | TreeMap cannot contain any null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. |

**Sorting in collections:**

**Collections** class provides static methods for sorting the elements of a collection. If collection elements are of a Set type, we can use TreeSet. However, we cannot sort the elements of List.

Collections.sort(list); //**sort()** is used to sort the elements of List, but List elements must be of the Comparable type.

 Collections.sort(list, Collections.reverseOrder()); //reverse order

**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.

Comparable & Comparator interfaces are used to order/sort the objects of the user-defined/custom classes.

**1)Comparable** --> It provides single sorted sequence i.e; you can sort the elements based on single data member only. It have compareTo() method only.

import java.util.\*;

import java.io.\*;

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) //use < for reverse order

return 1;

else

return -1;

}

}

public class TestSort3{

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);

}

}

}

**2)Comparator** ---> It provides multiple sorted sequences i.e., you can sort the elements on the basis of any/multiple data member and it have compare() and equals() methods.

class Student{

int rollno;

String name;

int age;

Student(int rollno,String name,int age){

this.rollno=rollno;

this.name=name;

this.age=age;

}

}

import java.util.\*;

class AgeComparator implements Comparator<Student>{ //age comparator

public int compare(Student s1,Student s2){

if(s1.age==s2.age)

return 0;

else if(s1.age>s2.age)

return 1;

else

return -1;

}

}

//Name comparator

class NameComparator implements Comparator<Student>{

public int compare(Student s1,Student s2){

return s1.name.compareTo(s2.name);

}

}

class TestComparator{

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));

System.out.println("Sorting by Name");

Collections.sort(al, new NameComparator());

for(Student st : al){

System.out.println(st.rollno+" "+st.name+" "+st.age);

}

System.out.println("sorting by Age");

Collections.sort(al, new AgeComparator());

for(Student st : al){

System.out.println(st.rollno+" "+st.name+" "+st.age);

|  |  |
| --- | --- |
| **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** | Comparator provides **compare() & equals() methods** |
| 4) Comparable is present in **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. |

} }

}

**Note:** Java 8 Comparator interface is a functional interface that contains only one abstract method.

# *Features of Java 8:*

1) Functional Interface

An Interface that contains only one abstract method is known as functional interface. It can have any number of default and static methods.

2) Lambda Expressions

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are like methods, but they do not need a name and they can be implemented right in the body of a method.

parameter -> expression

*(*parameter1*,* parameter2*)* -> { code block }

3) forEach() method

Java provides a new method forEach() to iterate the elements. A new, concise and interesting way to iterate over a collection. This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

**EX:** namesMap.forEach((key, value) -> System.out.println(key + " " + value));

list.forEach(name -> System.out.println(name));

**foreach() loop** **foreach() method**

Lambda operators is not used Lambda operator is used. lambdas are used and thus operations on variables outside the loop are not allowed, returns error.

It can be used to access arrays stream().foreach() Can access

and collections collections only

The return or control statements The return or control statements

work within the loop don’t allow within the loop but the

function calls are very easy to call.

for (String str : arr) { list.stream().forEach(s->{

System.out.print(str); System.out.print(s);

} };

list.forEach(System.out::println);

4) Java has introduced a new **Date and Time API** since Java 8. The java.time package contains Java 8 Date and Time classes

5) Default Methods

Java provides a facility to create default methods inside the **interface**. Methods which are defined inside the interface and tagged with default keyword are known as default methods. These methods are non-abstract methods and can have method body.

6) Streams Api

Java provides a new additional package in Java 8 called java.util.stream. Stream API is used to process collections of objects & allows you to perform operations on the elements like filtering, mapping, reducing, and sorting etc with concise and readable code.

**Features**:

* Stream does not store elements. It simply conveys elements from a source such as a data structure, array, or an I/O channel through a pipeline of computational operations.
* Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection **produces a new Stream** without the filtered elements, rather than removing elements from the source collection.
* Stream operations are evaluated lazily.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("A"))

.map(String::toUpperCase)

.collect(Collectors.toList());

7) Base64 encode and decode

Java provides a class Base64 to deal with **encryption**. You can encrypt and decrypt your data by using provided methods. You need to import java.util.Base64 in your source file to use its methods.

# *Java File I/O:*

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.

Java IO

**OutputStream:**

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

Java output stream hierarchy

import java.io.FileOutputStream;

public class FileOutputStreamExample {

public static void main(String args[]){

try{

FileOutputStream fout=new FileOutputStream("D:\\testout.txt");

String s="Welcome to javaTpoint";

byte b[]=s.getBytes(); //converting string into byte array

fout.write(b);

fout.close();

System.out.println("success...");

}catch(Exception e){System.out.println(e);}

}

}

**InputStream**

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

Java input stream hierarchy

import java.io.FileInputStream;

public class DataStreamExample {

public static void main(String args[]){

try{

FileInputStream fin=new FileInputStream("D:\\testout.txt");

int i=fin.read();

System.out.print((char)i);

fin.close();

}catch(Exception e){System.out.println(e);}

}

}

**FileWriter / FileReader:**

Java FileWriter class is used to write **character-oriented data** to a file. It is 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 method to write string directly.

package com.javatpoint;

import java.io.FileWriter;

public class FileWriterExample {

public static void main(String args[]){

try{

FileWriter fw=new FileWriter("D:\\testout.txt");

fw.write("Welcome to javaTpoint.");

fw.close();

}catch(Exception e){System.out.println(e);}

System.out.println("Success...");

}

}

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class. It is character-oriented class which is used for file handling in java.

package com.javatpoint;

import java.io.FileReader;

public class FileReaderExample {

public static void main(String args[])throws Exception{

FileReader fr=new FileReader("D:\\testout.txt");

int i;

while((i=fr.read())!=-1)

System.out.print((char)i);

fr.close();

}

}

# *Time/space complexity:*

**Time complexity** is defined as the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

There are different types of time complexities used, let’s see one by one:

**1. Constant time – O (1)**

**2. Linear time – O (n)**

**3. Logarithmic time – O (log n)**

**4. Quadratic time – O (n^2)**

**5. Cubic time – O (n^3)**

* **O(N log N)** – linearithmic time complexity. The time required to run the algorithm increases linearly with the size of the input, multiplied by the logarithm of the input size.
* **O(2^N)** – exponential time complexity. The time required to run the algorithm increases exponentially with the size of the input.

1. The addition of two scalar numbers requires one addition operation. the time complexity of this algorithm is constant: O(1)

2. Time complexity of a simple **loop** when the loop variable is **incremented or decremented** by a constant amount: O(n)

3. Time complexity of a **loop** when the loop variable is **divided or multiplied** by a constant amount: O(logn)

4. Time complexity of a **nested loop**: O(n^2)

5. Time complexities of different loops: O(m) + O(n)

**Space complexity** is the amount of memory required by the algorithm (no. of variables stored etc) to solve a problem. Lesser the space, faster the algorithm executes. It is also important to know that time and space complexity are not related to each other.

# *Important Questions:*

**1. map() vs flatMap()**

***map()*** operation is used to transform each element of a stream into a new stream using a given function. It returns the transformed elements in the same order as the original stream. This transformation is one-to-one, meaning each input element produces exactly one output element.

List<String> fruits = Arrays.asList("apple", "banana", "cherry");

List<Integer> lengths = fruits.stream()

.map(String::length)

.collect(Collectors.toList());//lengths: [5, 6, 6]

List number = Arrays.asList(2,3,4,5);

List square = number.stream().map(x->x\*x).collect(Collectors.toList());

**flatMap()** operation is used when each element in the stream is transformed into multiple elements, often in the form of another collection or stream. The resulting elements are then flattened into a single stream. This transformation is one-to-many, meaning an input element produces multiple output elements, later all flattened into a single Stream

ie;  flatMap() operation is a two-step process i.e. map() + flattening. means, it helps convert Collection<Collection<Item>> to Collection<Item>.

List<List<Integer>> listOfLists = Arrays.asList(

Arrays.asList(1, 2, 3),

Arrays.asList(4, 5),

Arrays.asList(6, 7, 8));

List<Integer> flattenedList = listOfLists.stream()

.flatMap(list -> list.stream())

.toList(); // [1, 2, 3, 4, 5, 6, 7, 8]

**2. Builder patterns:** The Builder pattern is a creational **design pattern** used to construct complex objects step by step. It aims to provide a **clear and readable way** to construct objects, avoiding the complexity of large constructors with many parameters.

- Types of Design patterns

1) Creational Design patterns: These patterns are designed for class instantiation or object creation – Singleton, Prototype, Factory Method, Abstract Factory, Builder, Object Pool.

2) Structural Design patterns: These patterns are about organizing different classes and objects to form larger structures and provide new functionality. The main goal of most of these patterns is to increase the functionality of the class(es) involved, without changing much of its composition - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data and Proxy.

3) Behavioral Design patterns: These patterns are designed depending on how one class communicates with others - Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor.

**3. CAP theorem, also known as Brewer's theorem**

**Consistency**: Consistency means that all clients see the same data at the same time, no matter which node they connect to. For this to happen, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed ‘successful.’

Ex: US bank balance and Ind bank bal should be same even if we withdraw at US

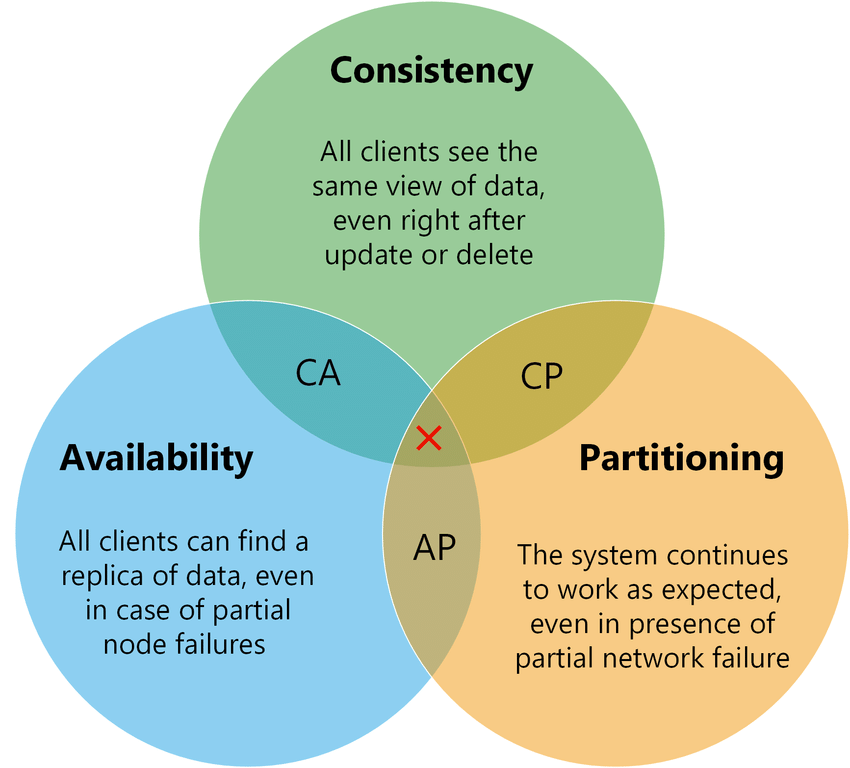
**Availability**: Availability means that any client making a request for data gets a response, even if one or more nodes are down.

Ex: Both indian and US can be able to view balance

**Patition Tolerance**: A partition is a communications break within a distributed system - a lost or temporarily delayed connection between two nodes. Partition tolerance means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.

Ex: Even if Us and Ind network didn't communicate, still services will work.

A distributed system can deliver atmost two of these three desired characteristics: consistency, availability, and partition tolerance.



4. **SOLID principles** -> The SOLID principle helps in reducing tight coupling.

Single Responsibility Principle

* This principle states that “a class should have only one reason to change” which means every class should have a **single responsibility** or single job or single purpose.

Open-Closed Principle

* This principle states that “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” which means you should be able to extend a class behavior, without modifying it.

Liskov’s Substitution Principle

* This principle states that “Derived or child classes must be substitutable for their base or parent classes” which means that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

Interface Segregation Principle

* This principle states that “do not force any client to implement an interface which is irrelevant to them“ which means larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

Dependency Inversion Principle

* This principle refers to the “decoupling of software modules”. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions. The main motive of this principle is decoupling the dependencies so if class A changes the class B doesn’t need to care or know about the changes.

5. **RLock in RWMutex package** vs **Lock in mutex package**

Lock: It will allow only single thread to lock and allows it to read or write.

RLock: It will allow configured or multiple threads to lock but only to read the data and not to write anything.

6. JSON Web Token (JWT) is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

Here are some scenarios where JSON Web Tokens are useful:

* **Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token.
* **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties.

In its compact form, JSON Web Tokens consist of **three** parts separated by dots (.), which are:

* Header
* Payload
* Signature

Ex: xxxxx.yyyyy.zzzzz

1)Header, It typically consists of two parts: 1)the type of the token, which is JWT. 2) the signing algorithm being used, such as HMAC SHA256 or RSA.

{

"alg": "HS256",

"typ": "JWT"

}

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

2) payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

Then, this JSON is **Base64Url** encoded to form the SECOND part of the JWT.

3)Signature To create the signature part, you have to take the encoded header, the encoded payload, a secret - the algorithm specified in the header, and sign that.

HMACSHA256(base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

The signature is used to verify that the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema.

Authorization: Bearer <token>

7. **static synchronization wrappers**

The method returns a thread-safe view of the **List**:

List<Integer> syncList = Collections.synchronizedList(new ArrayList<>());

The method returns a thread-safe view of the **Map**:

Map<Integer, String> syncMap = Collections.synchronizedMap(new HashMap<>());

The method returns a thread-safe view of the **SortedMap**:

Map<Integer, String> syncSortedMap = Collections.synchronizedSortedMap(new TreeMap<>());

The method returns a thread-safe view of the **Set**:

Set<Integer> syncSet = Collections.synchronizedSet(new HashSet<>());

The method returns a thread-safe view of the **SortedSet**:

SortedSet<Integer> syncSortedSet = Collections.synchronizedSortedSet(new TreeSet<>());

8. Sealed classes/Interfaces

A sealed class/Interface is a class that **explicitly specifies** which other classes are allowed to extend it. So, Add the “**sealed**” keyword to the class and specify which classes are permitted to inherit it by using the “**permits**” keyword.

*public* ***sealed*** *class Animal* ***permits*** *Dog, Cat, Bird {*

*//Class implementation*

*}*

In this example, the Animal class is declared as sealed and permits three subclasses: Dog, Cat and Bird. Any attempt to create a new subclass of Animal outside of this list will result in a **compilation error.**

**9. Difference between Btree and B+ tree:**

B-tree is known as a self-balancing tree (Binary Search Tree) as its nodes are sorted in the specific order, with the lowest value on the left and the highest value on the right. All the leaf nodes of the B-tree must be at the **same level** whereas, in the case of a *binary tree*, the leaf nodes can be at **different levels**.

B+ tree eliminates the drawback Btree used for indexing by storing data pointers only at the leaf nodes of the tree. B+ tree is also known as an advanced self-balanced tree. B+ tree is used to store the records very efficiently by storing the records in an indexed manner using the B+ tree indexed structure. Due to the multi-level indexing, the data accessing becomes faster and easier.

| **Basis of Comparison** | **B tree** | **B+ tree** |
| --- | --- | --- |
| **Pointers** | All internal and leaf nodes have data pointers | Only leaf nodes have data pointers |
| **Search** | Since all keys are not available at leaf, search often takes more time. | All keys are at leaf nodes, hence search is faster and more accurate. |
| **Redundant Keys** | No duplicate of keys is maintained | Duplicate of keys are maintained |
| **Insertion** | Insertion takes more time and it is not predictable sometimes. | Insertion is easier and the results are always the same. |
| **Deletion** | Deletion of the internal node is very complex, and the tree must undergo a lot of transformations. | Deletion of any node is easy because all nodes are found at leaf. |
| **Leaf Nodes** | Leaf nodes are not stored as structural linked list. | Leaf nodes are stored as structural linked list. |
| **Height** | For a particular number of nodes height is larger | Height is lesser than B tree for the same number of nodes |
| **Application** | B-Trees used in Databases, Search engines | B+ Trees used in Multilevel Indexing, Database indexing |
| **Number of Nodes** | Number of nodes at any intermediary level **‘n’ is 2n** | Each intermediary node can have **n/2 to n** children. |

**10. CountDownLatch  vs CyclicBarrier**

**CountDownLatch** is a thread waiting for multiple threads to finish or calling countDown(). When all threads have called countDown(), the awaiting thread continues to execute. (This is needed when we might need to start our application only when a particular set of tasks are completed)

In **CyclicBarrier** a group of threads waits together until all the threads arrive at a barrier. At that point, the barrier is broken, and action can optionally be taken.

**11. Volatile keyword**

The volatile keyword in Java is used to indicate that a variable's value may be modified by multiple threads simultaneously. It ensures that the variable is always read from and written to the main memory, rather than from thread-specific caches, ensuring visibility across threads. Volatile keyword is applicable with both primitive types and objects. When declared volatile, a variable does not cache its value and always reads from the main memory.

Ex: static **volatile** int sharedVar = 6; // volatile keyword here makes sure that the changes made in one thread are immediately reflect in other thread

**12. Enums**

An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).

To create an enum, use the enum keyword and separate the constants with a comma. Note that they should be in uppercase letters:

enum Level {

LOW,

MEDIUM,

HIGH

}

Level myVar = Level.MEDIUM; //access enum constants with the **dot** syntax

public enum MarketTypeEnum {

RETAIL("Retail"),

WHOLESALE("Wholesale");

private String marketType;

MarketTypeEnum(String marketType) {

this.marketType = marketType;

}

public String getMarketType(){ return marketType;}

}

**Note:** Enum declaration can be done outside a Class or inside a Class but not inside a Method.

* We can use ENUM in switch
* Loop through enum -> .values() gives all constant values as Array

13. **Shallow copy** vs **Deep copy**

Shallow Copy (default): In a shallow copy, a new object is created, but instead of copying the elements themselves, **references** to the original elements are copied. This means that changes made to the elements of the copied object will be reflected in the original object and vice versa.

In essence, a shallow copy creates a new collection or object, but it does not create copies of the elements within the collection; it simply copies the references to those elements.

Deep Copy: In a deep copy, both the object itself and all the objects contained within it are duplicated. This means that changes made to the elements of the copied object will not affect the original object, and vice versa. They are completely independent.

Deep copy creates entirely new instances of the objects and recursively copies all objects within the original object, ensuring that there are no shared references between the original and copied objects.

* To achieve this, made changes in @override clone() method of cloneable interface, by manually creating objects by copying all members.

Choosing between shallow and deep copy:

* If your objects contain only primitive data types or immutable objects, a shallow copy might be sufficient.
* If your objects contain mutable objects or you want to completely detach the copied object from the original, a deep copy is necessary.

# *Code practice:*

<https://takeuforward.org/interviews/strivers-sde-sheet-top-coding-interview-problems/>

https://github.com/fayaz-224/Java.git