**What is Java?**

* Java is a programming language and a platform.
* Java is an object-oriented programming language.
* Java is a high level and secure programming language.
* Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995.
* James Gosling is known as the father of Java. Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

**What is A high-level language?**

A high-level language is any programming language that enables development of a program in a much more user-friendly programming context

**What is programming language?**

A programming language is a computer language programmers use to develop software programs, scripts, or other sets of instructions for computers to execute.

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

**According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:**

* Desktop Applications such as acrobat reader, media player, antivirus, etc.
* Web Applications such as irctc.co.in, javatpoint.com, etc.
* Enterprise Applications such as banking applications.
* Mobile

**Types of Java Applications**

There are mainly 4 types of applications that can be created using Java programming

**1) Desktop applications or Standalone Application**

Standalone applications are also known as desktop applications or window-based applications

These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc.

**2)Web Application**

In Web application where user interacts

**3) Enterprise Application**

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application.

**4) Mobile Application**

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

**Java Platforms / Editions**

There are 4 platforms or editions of Java:

**1) Java SE (Java Standard Edition)**

It is a Java programming platform mostly developers will use

**2) Java EE (Java Enterprise Edition)**

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is used in companies

**3) Java ME (Java Micro Edition)**

It is a micro platform that is dedicated to mobile applications.

**4) JavaFX**

It is used to develop rich internet applications. It uses a lightweight user interface API.

**What is JVM?**

* JVM stands for Java Virtual Machine.
* It converts **Java bytecode into machines language**
* Java Virtual Machine (JVM) is a engine that provides runtime environment JRE to drive the Java Code or applications
* JVM is a part of **Java Runtime Environment (JRE)**

**The JVM performs the following main tasks:**

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

**Compiler:**

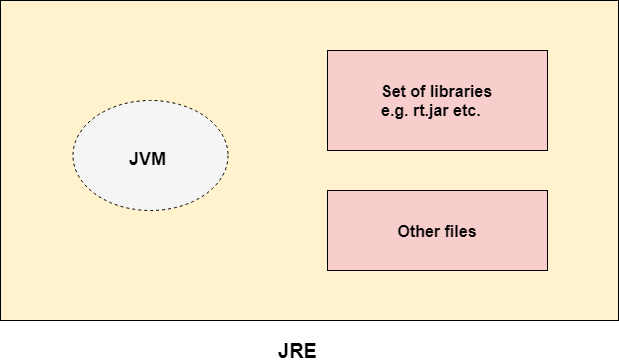
* compiler is software that converts a program written in a high-level language (Source Language) to a low-level language (Object/Target/Machine Language/0, 1’s).

**Interpreter:**

* Both compiler and interpreters do the same job which is converting higher level programming language to machine code.
* However, **a compiler will convert the code into machine code (create an exe) before program run**.
* Interpreters convert code into machine code when the program runs

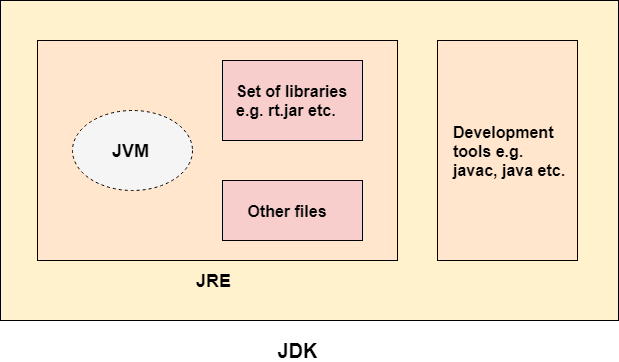
**What is JRE?**

* JRE stands for Java Runtime Environment.
* Using JRE, we can only execute already developed applications. We cannot develop new applications or modify existing applications.
* As the name suggests, JRE only provides Runtime Environment.



**What is JDK?**

* JDK stands for Java Development Kit.
* Using JDK, we can **develop, compile and execute (**run) **new applications** and also we can modify existing applications.
* We need to install JDK in developers machine where we want to develop new applications or modify existing applications.
* JDK includes JRE and development tools (environment to develop, debug and monitor Java programs)



**Machine language**

* Machine language is a collection of binary digits or bits that the computer reads and interprets.
* Machine language is the only language a computer is capable of understanding.

**Java Environment Setup:**

1Download and Install JAVA -Follow this site to install

<https://www.geeksforgeeks.org/how-to-download-and-install-java-for-64-bit-machine/>

**Comments in Java:**

In Java, we have two types of comments. We use comments to write some text within our code. Compiler will ignore these comments.

**1) Java Single Line Comment**

The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements.

**Syntax:**

1. //This is single line comment

**2) Java Multi Line Comment**

The multi-line comment is used to comment multiple lines of code.

Syntax:

1. /\*
2. This
3. is
4. multi line
5. comment
6. \*/

Shortcut Key for Eclipse :

Ctrl + Shift + /

**3) Java Documentation Comment**

Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API

**Syntax:**

/\*\*

\*

\*We can use various tags to depict the parameter

\*or heading or author name

\*We can also use HTML tags

\*

\*/

**Shortcut Key for Eclipse:**

**Shift-Alt-J** is a useful keyboard shortcut in Eclipse for creating Javadoc comment templates.

**What is Project?**

* A Java project contains source code and related files for building a Java program.

**Package:**

* A package in Java is used to group similar classes.
* Think of it as a folder in a file directory.
* We use packages to avoid name conflicts, and to write a better maintainable code.

**Example** – In your machine ,you have folder named with song which have all english,tamil,hindi songs if u want to play only english songs it is tough inroder to avoid we can create Three folders as English songs.Tamil Songs and Hindi songs

Similarly we can create packages which will have similar classes

Good practice to create package name as : com.lrj.learningjava (similar to Maven projects packages)

Always create class name in pascal case

**Pascal case examples**

* OutOfMemoryException
* DateFormat

**Class:**

* Class is a "**blueprint"** for creating objects.
* Everything in Java is associated with **classes and objects**, along with its **attributes** and **states**
* For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

**Class – Data + method (function)**

**Object – It is an instance of class**

**Method in Java:**

* In general, a method is a way to perform some task.
* A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.
* It provides the reusability of code
* We can also easily modify code using methods.



**Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.**Access Specifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:

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

Table

Description automatically generated

**Note**

- A subclass is a class that derives from another class.

**Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

**Method Name**: It is a unique name that is used to define the name of a method.

It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be subtraction(). A method is invoked by its name.

**Parameter List:**

It is the list of parameters separated by a comma and enclosed in the pair of parentheses.

It contains the data type and variable name.

If the method has no parameter, left the parentheses blank.

**Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

The most important method in Java is the **main()** method

**Java main() method**

The main() is the starting point for JVM to start execution of a Java program. Without the main() method, JVM will not execute the program.

Diagram

Description automatically generated

**The syntax of the main() method is:**

**public:** It is an access specifier. We should use a public keyword before the main() method so that JVM can identify the execution point of the program

**static:** You can make a method static by using the keyword static

**Rule for static :**

* We should call the main() method without creating an object.
* Static methods are the method which invokes without creating the objects, so we do not need any object to call the main() method.

**void**: In Java, every method has the return type. Void keyword acknowledges the compiler that main() method does not return any value.

**main():** It is a default signature which is predefined in the JVM.

It is called by JVM to execute a program line by line and end the execution after completion of this method.

We can also overload the main() method.

**String args[]:**

* The main() method also accepts some data from the user. It accepts a group of strings, which is called a string array.
* It is used to hold the command line arguments in the form of string values.

**Variable**

* A variable is the name of a reserved area allocated in memory



32.7M

703

C++ vs Java

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

Types of Variables

There are three types of variables in Java

1. local variable
2. instance variable
3. static variable

**1) Local Variable**

* A variable declared inside the body of the method is called local variable.
* You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

**2) Instance Variable**

* A variable declared inside the class but outside the body of the method, is called an instance variable.
* It is not declared as static

**3) Static variable**

* A variable that is declared as **static** is called a static variable. It cannot be local.
* You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory

**Example to understand the types of variables in java**

public class A

{

static int m=100;//static variable

void method()

{

int n=90;//local variable

}

public static void main(String args[])

{

int data=50;//instance variable

}

}//end of class

**Data Types in Java**

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

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

**Interfaces in java:**

* An interface is a **blueprint of a class.**
* Like a class, **an interface can have methods and variables,** but the methods declared in an interface are by **default abstract** (**only method signature, no body**).
* Rules :Variable in interface **are public, static and final**
* **All interface variables are public in nature**
* It is used to provide **total abstraction 100%**
* If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
* To declare an interface, use **interface** keyword

**Why do we use interface ?**

* It is used to achieve **total abstraction.**
* Since java does not **support multiple inheritance** , but by using interface we can achieve multiple inheritance .
* It is also used to **achieve loose coupling.**

Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes? The reason is, abstract classes may contain non-final variables, whereas variables in interface are public, static and final and multiple inheritance is possible only in interface

In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

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

* **Abstract Class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

**Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

**Difference between Abstract Class and Interface**

**Abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).**

|  |  |
| --- | --- |
| **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) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 5) An **abstract class** can extend another Java class | An **interface** can extend another Java interface only. |
| 6) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 7) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

**Collection in Java:**

* Collection in Java is a framework that used to **store and manipulate the group** of object

**Collection in Laymen term:**

* **Collection a is group of Individual objects**
* **Example : A library have collection of books**
* **Music album will have collection of songs**

A diagram of a computer program

Description automatically generated with medium confidence

**Arrays:**

* Collection of similar elements is known as Array.
* Array have Fixed length
* Array in Java is an Object
* Array object holds fixed number of values of a similar data types which means an array of int will contain only integers, an array of string will contain only strings etc..
* The length of an array is established when the array is created. After creation, its length is fixed. Array is a index based and its index starts from 0 which means the first element of an array is stored at 0 index. Array holds primitive types as well as object references

**List**

* List maintains **Ordered collection**
* List allows duplicate values
* Java List is an **Interface ,** which **inherits the Collection interface.**
* **The implementation classes** of List interface are **ArrayList and LinkedList**

**List Interface declaration**

public interface List extends Collection

**How to create List**

//Creating a List of type String using ArrayList

List<String> list=new ArrayList<String>();

**//Creating a List of type Integer using ArrayList**

List<Integer> list=new ArrayList<Integer>();

**ArrayList:**

* It is resizeable array
* **Maintains Ordered collection**
* Duplicate values are allowed
* Arraylist is non synchronized, So it is NOT thread safe
* Array list implemented using LIST interface

Array list Syntax:

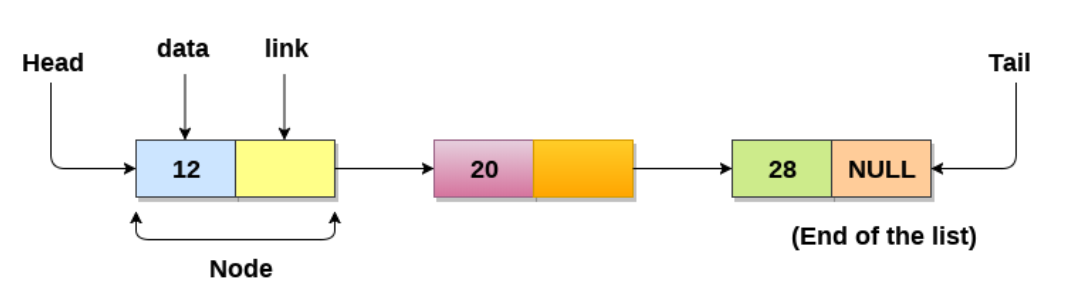
ArrayList<String> arm = new ArrayList<String>();

**Linked List:**

* **LinkedList uses a doubly linked list to store the elements.**
* Linked List used for dynamic memory allocation
* Efficient in insertion and deletion operations compared to arrays.
* LinkedList Implemented Using List Interface and Deque Interface

**Single Linked List:**

* A node contains two fields
* One field contains data and another field has a reference (link) to the next node
* Nodes are randomly stored in the memory.



**Doubly Linked List:**

* Doubly Linked List has 3 fields
* First Field has reference links pointers for the previous nodes
* Second Field has data
* Third Field has reference links for the Next Node

**Advantages of Linked List:**

* Java LinkedList class is **non synchronized.**
* Duplicate values are allowed
* Maintains insertion order
* Manipulation is fast

LinkedList<String> li = **new** LinkedList<String>(); //Creating an linkedlist object

li.add("john"); //Adding an element

li.remove("john"); //removing an element

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList uses a **dynamic array** to store the elements. | LinkedList uses a **doubly linked list** to store the elements. |
| 2) Manipulation with is **slow** . | Manipulation is **faster** |
| 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) Memory location of an **ArrayList is contiguous**. | Memory location **of linked list is not contagious.** |
| 6) when an ArrayList is initialized, a default capacity of 10 arrays is assigned to the ArrayList. | when an LinkedList is initialized an empty list is created ,So there is No a default capacity of arrays in linked list |
| 7) To be precise, an ArrayList is a resizable array. | LinkedList implements the doubly linked list of the list interface. |

**HashSet**

* **HashSet is a part of collection that a hash table for storage.**
* Example - Whenever we add any element to HashSet, it puts element into backing HashMap with key as our element and value as static field PRESENT.
* Hash Set inherits the AbstractSet class and implements Set interface.

A table with text on it

Description automatically generated

**The important points about Java HashSet class are:**

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains **unique elements only(No duplicates element).**
* HashSet **allows null value.**
* HashSet class is **non synchronized.**
* HashSet **doesn't maintain the insertion order.**

**Understanding HashSet:**

**public** **class** HashSet<E>

**extends** AbstractSet<E>

**implements** Set<E>, Cloneable, java.io.Serializable

{

**static** **final** **long** serialVersionUID = -5024744406713321676L;

**private** **transient** HashMap<E,Object> map;

    // Dummy value to associate with an Object in the backing Map

**private** **static** **final** Object PRESENT = **new** Object();

    /\*\*

     \* Constructs a new, empty set; the backing <tt>HashMap</tt> instance has

     \* default initial capacity (16) and load factor (0.75).

     \*/

**public** HashSet() {

        map = **new** HashMap<E,Object>();

    }

   // Other constructors and methods

}

* there is one static field PRESENT of type Object
* Whenever we add any element to HashSet, it puts element into backing HashMap with key as our element and value as static field PRESENT.

For Example,

import java.util.HashSet;

public class HashSetInternalDemo{

public static void main(String[] a){

HashSet<String> friendList = new HashSet<>();

friendList.add("Malay");

friendList.add("Mahesh");

friendList.add("Jay");

}

}

In above java program, I have created HashSet < String > and added 3 different Strings in it. Internally it creates HashMap < String, Object> and stores elements as shown below.

A table with text on it

Description automatically generated

Source: https://codepumpkin.com/hashset-internal-implementation/

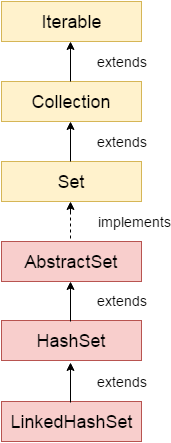


**Difference between List and Set**

* A list can contain duplicate elements whereas Set contains unique elements only.

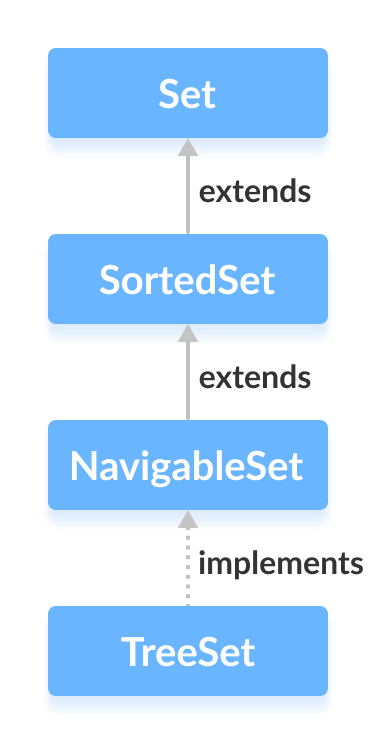
**LinkedHashSet**

* LinkedHashSet class is a combination of Hash Table and Linked list implementation of the Set interface.
* It inherits the HashSet class and implements the Set interface.
* Java LinkedHashSet class is **non-synchronized.**
* Java LinkedHashSet class **maintains insertion order**.



**Java Tree Set**

* Collections framework provides the functionality of a **tree data structure.**
* It uses a tree data structure to store and maintain the objects.
* It extends the NavigableSet
* implements the Set interface



**Java Hashtable class**

* Hashtable stores **key and value in pairs**
* It is similar to HashMap, but is synchronized.So thread safe
* Java Hashtable class contains **unique elements.**
* Java Hashtable class **doesn't allow null value.**
* Hastable is slow and legacy system
* It inherits Dictionary class and implements the Map interface

A screenshot of a computer

Description automatically generated

**Declaration:**

public class Hashtable<K,V> extends Dictionary<K,V> implements Map

**Type Parameters:**

K – the type of **keys** maintained by this map

V – the type of mapped **values**

**Java Hash Map:**

* HashMap stores **Key and value in pairs**
* HashMap class implements the **Map interface**
* Keys should be unique. If you try to insert the duplicate key, it will replace the element into corresponding original key
* It is easy to perform operations using the **key index** like updation, deletion, etc.
* Hashmap is fast and introduced in JDK 1.2

Example :World cup years with the country which won it.

|  |  |  |
| --- | --- | --- |
| **Slno** | **Key** | **Value** |
| 1 | 1983 | India |
| 2 | 1987 | Australia |
| 3 | 2003 | Australia |
| 4 | 2011 | India |
| 5 | 2023 | India |
|  |  |  |
|  |  |  |

A diagram of a key value pair

Description automatically generated

Synaax:

HashMap< Integer, String> map = **new** HashMap<>();

Methods : Add an element using PUT

Remove an element using Remove method

Size method for the length measurement

Absentif

**Points to remember**

* Java HashMap contains values based on the key.
* Java HashMap maintains no insertion order.
* Java HashMap is non synchronized.

**Java LinkedHashMap**

* LinkedHashMap is just like HashMap with an additional feature of maintaining an order of elements inserted into it.
* HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order..

**Points to remember**

* Java LinkedHashMap contains values based on the key.
* Java LinkedHashMap contains unique elements.
* Java LinkedHashMap is non synchronized.
* Java LinkedHashMap maintains insertion order.



**TreeMap class**

* TreeMap class is **a red-black tree** based implementation
* It provides an efficient means of **storing key-value pairs in sorted order.**

**Points to remember:**

* Java TreeMap contains values based on the key.
* Java TreeMap contains only unique elements
* Java TreeMap is non synchronized.
* Java TreeMap maintains ascending order.

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe. | Hashtable is **synchronized**. It is thread-safe |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 6) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 7) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

**Scanner**

* Scanner class allows us to **read** input values from the user.
* In other worlds, by using Scanner, we can **ask** the input values to the user of the program

**Iterator**

* **Java Iterator** is an interface that is used to iterate over **a collection of object one by one**

Eg Collection of java object like arraylist

It has 4 methods:

* hasNext()
* next()
* remove()
* forEachRemaining()

**String:**

* String is an object that represents sequence of char values
* Eg : “AGNI” is a string of 4 characters
* Sting is an immutable object which means it is constant and cannot be changed once it is created
* The Java.lang.String class implements Serializable, Comparable ad CharSequence Interfaces
* CharSequence Interfaces implemented by String,String Bufferand String Builder

**How to Create a String?**

1.By String Literal

2.By new keyword

1.By String Literal

String string1 = “Agni”;

String sting2 = “Agni”;

* First string will check for Agni in heap memory,If Agni was not there,it will create instances of Agni
* If string String2 try to create Agni,it will check for Agni,Here Agni is already created so it will refer or points the Agni
* Memory efficiency is high for By string literal

2.By new keyword

String s = new String (“Agni”)

* It will create new instance of Agni
* Each time new separate object is created here it wont refer like String literal ,So memory efficiency is low

**String is Immutable? How**

Create a String name = “Nazeer:

String name = “Alex”

So the output will be Alex,as we thing it is override

But actually name is just point to Alex ,So string name = Nazeer will be there,it is ready for garbage collection

**String Buffer:**

* An object that represents sequence of char values
* In java ,StringBuffer is a mutable object
* String Buffer is synchronized and Thread-Safe
* We can create object only using new keyword

**Why Stringbuffer is Mutable and String is Immutable?**

* If we give string name as Nazeer ,and Add my firstname using concate method it will print as Nazeermohamed but again if you pring strint name it will show as Nazeer
* Whereas in Stringbuffer if you give my name as Nazeer using append method added my first name as Mohamed,then it will give Nazeer Mohamed
* But again ifyou print the String name it will change to Nazeer Mohamed so we call it as Immutable

**Can we reverse a string?**

String cannot be reversed,But Stringbuffer can be reversed using reverse function

**String Buffer methods:**

* Reverse
* Replace
* Delete
* Insert
* Capacity

And also String method like charAt, Substring ,length methods can be used in stringbuffer

And also Stringbuffer methods can be used in Stringbuilder methods

**String Builder**

* An object that represents sequence of char values
* StringBuilder is a mutable object
* StringBuilder class is a child of charSequence
* StringBuilder is **Non-Synchronised / NOT thread-safe(i.e Multiple threads can be accessed)**
* **String builder can use Stringbuffer methods**

**Diff Between String Buffer vs Sting Builder**

* StringBuffer is synchronized which means two thread cant call the method simultaneously and Stringbuilder is non-synchronized which means tow threads can call the methods simultaneously
* String buffer is less efficient
* String builder is more efficient

**Wrapper Class:**

* The Wrapper class is a mechanism to convert primitive data types into object and Object into primitive data types
* Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects):

Wrapper classes in java:

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) |
| char | [Character](https://www.javatpoint.com/post/java-character) |
| byte | [Byte](https://www.javatpoint.com/java-byte) |
| short | [Short](https://www.javatpoint.com/java-short) |
| int | [Integer](https://www.javatpoint.com/java-integer) |
| long | [Long](https://www.javatpoint.com/java-long) |
| float | [Float](https://www.javatpoint.com/java-float) |
| double | [Double](https://www.javatpoint.com/java-double) |
|  |  |

**Autoboxing**

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing,

for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

**Unboxing**

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing

**Exception** An unexpected event occurs during execution of a program is called **Exception**

**Exception Handling:**

* If the normal flow of the program’s Instruction get disrupt, if it is handled by our program which is called **Exception Handling**

**An Exception is NOT an ERROR**

* An Error is a very severe problem, which can’t be handled by the program
* Example – When your computer runs OUT OF SPACE error, there is nothing program can do to stop the program crashing

**Who is dangerous?**

* Unchecked exception are dangerous because it is not known until the program runs whereas checked exception indicated by compiler itself so it is danger

**Checked exceptions** are checked at compile-time and must be either

caught or declared in the method signature using 'throws.

Example – We caught and handle this exception using Try ,Catch and Finally block

**Unchecked exceptions** are not checked at compile-time.it will get caught only when the program runs.So only it also called as RuntimeExceptions

// Checked exception (must be handled or

declared)

try {

FileInputStream file = new FileInputStream("file.txt");

} catch (FileNotFoundException e) {

System.out.println("File not found.");

}

// Unchecked exception (no need to declare or

catch)

int result = 10 / 0; // ArithmeticException

**Try Block** :

This is the block where we write our code with assumptions that it may give exception

'Try' is used to enclose code that might throw an exception

**Catch Block**: This is the block where exception are handled. Catch block is immediately followed by try

'catch' is used to handle exceptions

object

Assertion Error etc

Virtual Machine Error

Unchecked Exceptions Eg Runtime or Null exception

Checked Exception eg IO or Complete time exception

Exceptions

Error

Throwablee

**Finally:**

finally' is used to specify code that will always execute, whether an exception occurs or not

Example for Try,Catch and Finally:

try

{      // Code that might throw an exception      int result = 10 / 0; }

catch (ArithmeticException e)

{      // Handle the exception         System.out.println("Error: " + e.getMessage()); }

finally {      // Cleanup code (always executed)

   System.out.println("Cleanup code"); }

**Important Points:**

* A try block can have multiple catch block
* If there are multiple catch blocks should not be of Generic Exception type
* If there are no exceptions, then the catch blocks will not get executed.

**Finally Blocks:**

* This is a block of code which will get executed even if the exception occurs or not
* This block will contain code like closing connection of DBs,buffer,streams etc

**Important Points:**

* There is no finally block without a TRY-CATCH block. Finally block must be associated with a try catch
* Finally block is not mandatory.
* Finally block will be executed after CATCH block,If there is a exception
* Otherwise Finally block will be executed after try block
* Finally block also can have exception
* Finally block will execute even if the try block contains statements like return, break or continue

**Will Finally block will always execute?**

* Oneirically will not executed if

1.The thread is dead

2.When System. Exit() is called

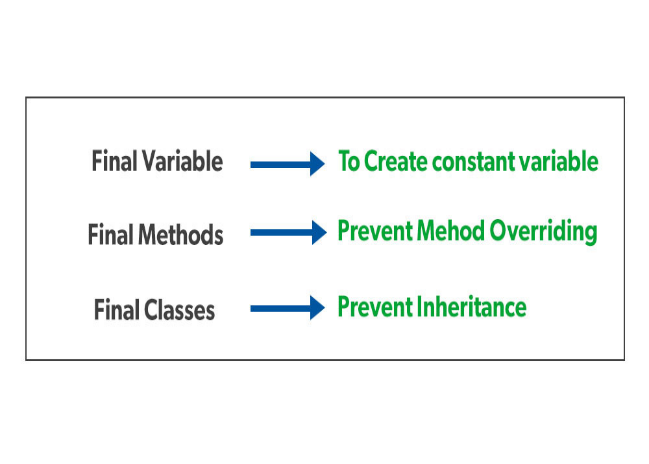
3.When an unrecoverable exception happens in Finally block

**What is JRE? JRE provides the libraries**

What is JDK?JDK contains JRE,Compiler,Debugger the Java Virtual Machine (JVM), and other components to run applets and applications written in the Java programming language.

**What is Final Keyword?**

* Final keyword is a non-access modifier used for classes, variables and methods, which makes them **non-changeable**
* Final keyword in Java has three different uses: create constants, prevent inheritance and prevent methods from being overridden.



Java Program to check the string is Palindrome:

What are all methods used in windows handling?

Getwindowhandle

Getwidnowhandles

**Super Keyword in java:**

**Usage of Java super Keyword**

* super can be used to **refer**  parent class **instance variable.**
* super can be used to **refer** parent class **method.**
* super() can be used to **refer** parent class **constructor.**

**Why we need super keyword?**

* Most common use of the super keyword is to eliminate the confusion between superclasses and subclasses that **have methods with the same name.**

Source: <https://www.programiz.com/java-programming/super-keyword>

**Multi-threading:**

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

**Advantages of Java Multithreading**

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

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

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

**Uses of Multi-threading:**

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

**Thread in java**

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

**Multitasking**

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

1. Process-based Multitasking (Multiprocessing)
2. Thread-based Multitasking (Multithreading)

**1) Process-based Multitasking (Multiprocessing)**

* Each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.

**2) Thread-based Multitasking (Multithreading)**

* Threads share the same memory area.
* A thread is lightweight.
* Cost of communication between the thread is low.

**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.

**Throw:**

* Throw keyword is used to create a custom error message.

**How it should be used**

* throw keyword is used together with an exception type
* Throw keyword should be used only once in a method
* Used on inside the method
* There are many exception types available in Java: ArithmeticException, ClassNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc.

public class LearnThrowkeyword {

public void accessfinder(int age) {

if(age>=18) {

System.out.println("Access granted");

}

else {

throw new ArithmeticException("Access denied"); //ArithmeticException is the exception type

//Access denies is the custom error message

}

}

public static void main(String[] args) {

LearnThrowkeyword lr = new LearnThrowkeyword();

lr.accessfinder(15);

**Throws:**

* Used to indicate what exception type should be thrown by a method
* Can declare multiple times
* Throws used in class and in method signature

|  |  |
| --- | --- |
| **throw** | **throws** |
| Used to throw an exception for a method | Used to indicate what exception type may be thrown by a method |
| Cannot throw multiple exceptions | Can declare multiple exceptions |
| **Syntax:**   * throw is followed by an object (new *type*) * **used inside the method** | **Syntax:**   * throws is followed by a class * **and used with the method signature** |
|  |  |

**File handling in Java :**

**Source:** **https://www.geeksforgeeks.org/file-handling-in-java/**

* file handling means reading and writing data to a file
* With the help of File Class, we can work with files.
* This File Class is inside the java.io package.
* The File class can be used by creating an object of the class and then specifying the name of the file.

**Streams in Java**

* In Java, a sequence of data is known as a stream.
* This concept is used to perform I/O operations on a file.

There are two types of streams :

**1. Input Stream:**

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

Some important subclasses of the InputStream class, which are as follows:

* FileInputStream
* ObjectInputStream
* AudioInputStream
* FilterInputStream

Creating an InputStream

// Creating an InputStream

InputStream obj = new FileInputStream();

**Methods of InputStream**

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

**2. Output Stream:**

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

Some Important subclasses are:

* FileOutputStream
* ObjectOutputStream
* PrintStream

Creating an OutputStream

// Creating an OutputStream

OutputStream obj = new FileOutputStream();

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

**Constructor Overloading in Java:**

* In Java, we can overload constructors like methods.
* Constructor overloading is defined as the concept of having more than one constructor with different parameters.

**SOLID principles** in Java (or any object-oriented language) are a set of five design principles aimed at creating clean, maintainable, and scalable software.

The SOLID acronym stands for:

**✅ 1. S – Single Responsibility Principle (SRP)**

**A class should have only one reason to change.**

* Each class should focus on a single task or responsibility.

**Example:**

class Invoice {

public void calculateTotal() { /\* logic \*/ }

}

class InvoicePrinter {

public void printInvoice(Invoice invoice) { /\* logic \*/ }

}

Instead of putting both logic and printing in the same class, responsibilities are separated.

### ✅ 2. **O – Open/Closed Principle (OCP)**

**Software entities should be open for extension but closed for modification.**

**⚠️ Problem Without OCP**

**Let's say you're calculating area for different shapes.**

**java**

**CopyEdit**

**class AreaCalculator {**

**public double calculateArea(Object shape) {**

**if (shape instanceof Circle) {**

**Circle c = (Circle) shape;**

**return Math.PI \* c.radius \* c.radius;**

**} else if (shape instanceof Rectangle) {**

**Rectangle r = (Rectangle) shape;**

**return r.length \* r.width;**

**}**

**return 0;**

**}**

**}**

* **Every time you add a new shape (like Triangle), you must modify the calculateArea() method.**
* **This violates OCP – the class is not closed for modification.**

**✅ OCP-Compliant Version**

**Let’s use polymorphism and abstraction:**

**java**

**CopyEdit**

**interface Shape {**

**double area();**

**}**

**class Circle implements Shape {**

**double radius;**

**Circle(double radius) { this.radius = radius; }**

**public double area() {**

**return Math.PI \* radius \* radius;**

**}**

**}**

**class Rectangle implements Shape {**

**double length, width;**

**Rectangle(double length, double width) {**

**this.length = length;**

**this.width = width;**

**}**

**public double area() {**

**return length \* width;**

**}**

**}**

**class AreaCalculator {**

**public double calculateArea(Shape shape) {**

**return shape.area();**

**}**

**}**

**Now if you add a new shape:**

**java**

**CopyEdit**

**class Triangle implements Shape {**

**double base, height;**

**Triangle(double base, double height) {**

**this.base = base;**

**this.height = height;**

**}**

**public double area() {**

**return 0.5 \* base \* height;**

**}**

**}**

**✅ You don't need to modify AreaCalculator.  
✅ You just extend by creating a new class.**

✅ 3. **L – Liskov Substitution Principle (LSP)**

* A child class should not break the behavior of the parent class

class Bird {

public void fly() {}

}

class Sparrow extends Bird {}

class Ostrich extends Bird {

@Override

public void fly() {

throw new UnsupportedOperationException("Ostrich can't fly");

}

}❌ This violates LSP because Ostrich cannot be used in place of Bird.

✅ A better design:

class Bird {}

class FlyingBird extends Bird {

public void fly() {}

}

class Sparrow extends FlyingBird {}

class Ostrich extends Bird {}

✅ 4. **I – Interface Segregation Principle (ISP)**

* Prefer small, specific interfaces over large, general-purpose ones.

**🔴 Problem Without ISP: Fat Interface**

**java**

**CopyEdit**

**interface Worker {**

**void work();**

**void eat();**

**}**

**Now if we create two classes:**

**java**

**CopyEdit**

**class HumanWorker implements Worker {**

**public void work() { System.out.println("Working..."); }**

**public void eat() { System.out.println("Eating lunch..."); }**

**}**

**class RobotWorker implements Worker {**

**public void work() { System.out.println("Robot working..."); }**

**public void eat() {**

**// Robot doesn't eat — violation of ISP**

**throw new UnsupportedOperationException("Robot doesn't eat");**

**}**

**}**

* **RobotWorker is forced to implement eat() even though it makes no sense. This violates ISP.**

**✅ Correct Design With ISP**

**Split the interfaces:**

**interface Workable {**

**void work();**

**}**

**interface Eatable {**

**void eat();**

**}**

**Now classes implement only what they need:**

**java**

**CopyEdit**

**class HumanWorker implements Workable, Eatable {**

**public void work() { System.out.println("Human working..."); }**

**public void eat() { System.out.println("Eating lunch..."); }**

**}**

**class RobotWorker implements Workable {**

**public void work() { System.out.println("Robot working..."); }**

**}**

* **Now each class only implements methods it actually needs.**
* **RobotWorker is not forced to implement eat() — this respects ISP.**

}

class HumanWorker implements Workable, Eatable {}

class RobotWorker implements Workable {}

✅ 5. **D – Dependency Inversion Principle (DIP)**

* High-level modules should not depend on low-level modules. Both should depend on abstractions or Interfaces
* Use interfaces or abstract classes so that high-level modules aren’t tightly coupled to low-level modules.

**🧑‍💻 Java Example (Before and After DIP)**

**❌ Without DIP**

java

CopyEdit

class MySQLDatabase {

public void save(String data) {

System.out.println("Saving data to MySQL: " + data);

}

}

class OrderService {

MySQLDatabase db = new MySQLDatabase();

public void placeOrder(String order) {

db.save(order);

}

}

* OrderService depends directly on MySQLDatabase. It's tightly coupled.

**✅ With DIP**

java

CopyEdit

interface Database {

void save(String data);

}

class MySQLDatabase implements Database {

public void save(String data) {

System.out.println("Saving data to MySQL: " + data);

}

}

class OrderService {

private final Database db;

public OrderService(Database db) {

this.db = db;

}

public void placeOrder(String order) {

db.save(order);

}

}

* Now, OrderService depends on the Database interface.
* You can pass any implementation (MySQLDatabase, MongoDBDatabase, etc.).
* Ideal for **Dependency Injection** frameworks like Spring.

**Summary of SOLID**

| **Principle** | **Focus** | **Helps With** |
| --- | --- | --- |
| SRP | One responsibility per class | Maintainability |
| OCP | Extend, don’t modify | Scalability, robustness |
| LSP | Subclasses should behave like base classes | Correctness |
| ISP | Interface size matters | Flexibility, decoupling |
| DIP | Depend on abstractions | Loose coupling |
| -------------------------------- |  |  |

**Design Pattern:**

* Design Patterns is a solutions to commonly occurring problems in software design.

1. **Creational patterns**

* These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

**Factory Method** is a creational design pattern that **provides an** **interface for creating objects in a superclass**, **but allows subclasses to alter the type of objects(override) that will be created.**

**Notification - interface**

**Emailnotification - concrete class**

**SmSnotification - concrete class**

**Notificationfactory - superclass**

**Emailnotificationfactory - subclass**

**SMSnotificationfactory - subclass**

 Notification is the **interface** (the product).

 EmailNotification and SMSNotification are **concrete products**.

 NotificationFactory is the **creator (superclass)** with a factory method.

 Subclasses like EmailNotificationFactory or SMSNotificationFactory **override the factory method** to create specific objects.

**🎯 Why use Factory Method?**

* You don't want the client code to know the exact class of the object.
* You want to make it easier to add new types without touching existing logic (Open/Closed Principle

| **Benefit** | **Description** |
| --- | --- |
| **Encapsulation of object creation** | Hides the logic of which object is created |
| **Loose coupling** | Client doesn't need to know concrete class |
| **Scalability** | Easy to add new types without changing core logic |

**🔧 Real-world Analogy**

Think of a **Pizza Store**. When a customer orders a pizza, they don’t care *how* it's made — they just want the final product.

The store (factory) decides *which pizza class* (e.g., CheesePizza, VegPizza) to instantiate and return.

Source: <https://refactoring.guru/design-patterns/factory-method>

**Singleton Design Pattern:**

* **Singleton Pattern** is a **creational design pattern** that ensures **only one instance** of a class is created and provides a **global point of access** to that instance.

**✅ Key Concepts**

1. **Private constructor** – Prevents direct instantiation.
2. **Static instance** – Holds the one and only instance.
3. **Public static method** – Returns the instance.

**Real World Example : A country will have only one president**

A group of men in suits

AI-generated content may be incorrect.

**What are all the design patterns used in your project?**

Cucumber Project Uses below Pattersn

**1. Page Object Model**

* You're calling methods from CheckoutPage:

* This encapsulates logic for interacting with web elements.

**2. Dependency Injection**

* The Checkoutstepdefinition constructor receives a TestContextSetup object:

* This pattern:
  + Promotes test reusability and scalability
  + Avoids tight coupling between step definitions and object instances

**3. Facade Pattern**

* Through PageObjectManager:
* This hides complexities and provides a unified method to access all page objects.

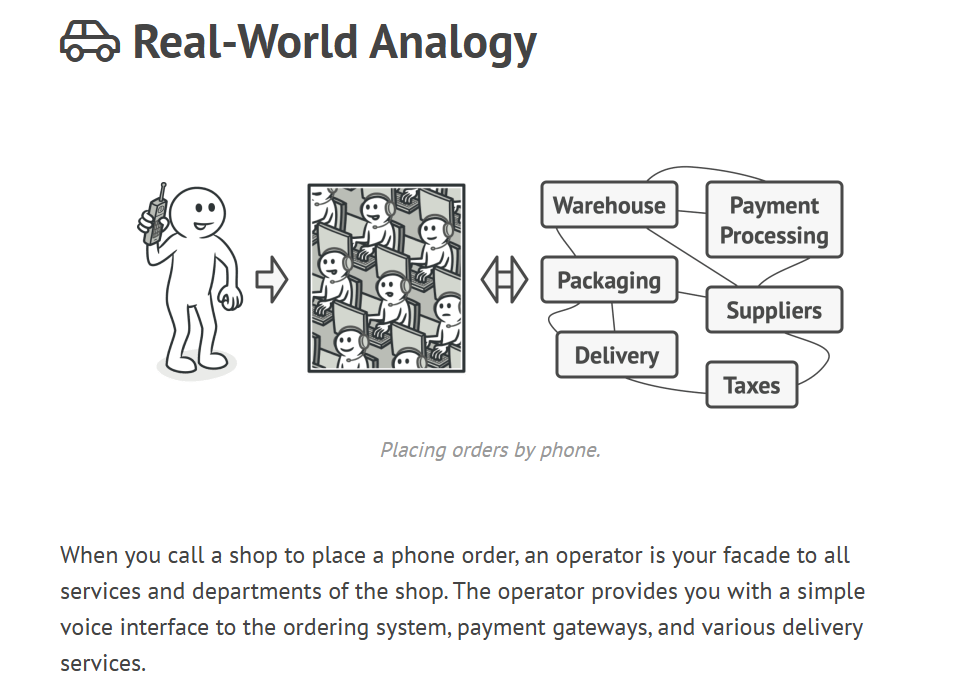
javaCopyEdittestContextSetup.pageObjectManager.getCheckoutpage();

javaCopyEditpublic Checkoutstepdefinition(TestContextSetup testContextSetup)

javaCopyEditcheckoutPage.selectMensCasualProduct();  
checkoutPage.selectaddtocart();  
checkoutPage.selectcheckout();  
 =========================

What is **Façade Design pattern?**

* Facade is a structural design pattern that provides a **simplified interface to a library, a framework**, **or set of classes.**



**Customized Example :** A Customer in a hotel room , call the reception for room service, laundry and other departments.Here reception act as simplified interface