|  |  |  |  |
| --- | --- | --- | --- |
| CALLER METHOD | CALLED METHOD | **Calling WAY**  With in same class | **Calling Way**  With in the different class |
| static | static | Directly | By class name |
| static | Non-static | By object reference variable Name | By object reference variable Name |
| Non-static | static | Directly | By class name |
| Non-static | Non-static | Directly | By object reference variable Name |

How we can call methods?

**Java Definiti**on :

* Java is a high-level programming language used to represent or correlate real-world entities.
* Java was developed by Sun Microsystems, with James Gosling as the lead developer, in 1995.

**Program :**

Program is a set of instruction which is use to perform specific operation.

Programing language:

The langue which is use to create a set of instruction.

Type of Language:

* Low level Language : which is understandable by machine. (Exa Binary code)
* Middle level Language : Which is minimum understand by human.(Exa byte code)
* High level Language : Which is understandable by human.(Exa java, Paython, C++, C#, C)

A screenshot of a computer screen

Description automatically generated

**Java Features:**

1. Platform Independence: Write Once, Run Anywhere" (WORA) via JVM.
2. Object-Oriente: Supports OOP principles: Encapsulation, Inheritance, Polymorphism, Abstraction.
3. Robust: Strong memory management, exception handling, and garbage collection.
4. Secure: Built-in security features (e.g., security manager, bytecode verification).
5. Multi-threadin: Supports concurrent execution of multiple threads for better performance.
6. High Performance: JIT compiler improves execution speed.
7. Dynamic: Can load classes dynamically and supports reflection.
8. Rich Standard Librar: Extensive APIs for data structures, I/O, networking, and GUIs.
9. Automatic Memory Management: Garbage collection eliminates memory management issues.
10. High-Level Language: Abstracts low-level details, making development easier.
11. Scalability: Suitable for both small and large-scale applications.

**#Diff b/w JDK, JRE and JVM?**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | JDK (Java Development Kit) | JRE (Java Runtime Environment) | JVM (Java Virtual Machine) |
| Purpose | To develop Java applications | To run Java applications | To execute Java bytecode |
| Components | Compiler, debugger, JRE | JVM and class libraries | Core engine for execution |
| Platform Dependency | Platform-dependent | Platform-dependent | Platform-independent |
| Relationship | JDK includes JRE | JRE includes JVM | JVM is the core component of JRE |
| Use Cases | Developers | End-users | Underlying execution engine |

**JDK**

**JRE (SET OF LIBRARIES)**

**Execution process :**

* A developer writes the code and saves it as a Java file, which is in a high-level language.
* Then, the file is sent for compilation, where it may result in either a compilation error or success.
* If the compilation is successful, the code is converted into bytecode.
* The bytecode is then executed by the JVM (Java Virtual Machine),
* which converts it into binary code. Finally, the result is displayed on the monitor.

#**Execution Process :**

**A screenshot of a computer

Description automatically generated**

**Conclusion**

* When a Java program is executed, two memory areas are created:
  + Stack Area
  + Heap Area
* Stack Area is used for execution purposes.
* Heap Area is used for storage purposes.
* JVM makes use to call resources:
  + Class loader
  + Main method
  + Garbage collector
* Every operation needs to be executed in the stack area.
* Class loader is responsible to create the static pool area and load all the static methods and variable into the static pool area.
* For each class, class loader will come and load the classes once.
* The new operator is responsible to create objects and load all the non static method and variable to the heap area, this will pointing to static pool area to the won class.
* Object address is assign to the object reference variable.
* After execution all the heap area clean by the garbage collector.

**#Object Class**

* **Object class** is an inbuilt class present in the java.lang package.
* **Object class** is the super most class present in Java.
* **Object class** contains common methods which will be applicable in any type of object, so that the name they have given it is Object.
* In between our class and Object class, there is an implicit inheritance.
* When a developer needs to perform inheritance, the compiler itself will be performing the inheritance.
* Object class contains **11 common methods** which will be applicable in all the classes present in Java.
* All the object class methods are **non-static methods**.
* While performing inheritance, we have to create an object of the subclass. By using the subclass object reference variable, we are going to access object class properties.
* Object class properties follow **1st method** as:

**Object class methods:**

1. **toString(): String**
   * It is non static method, in entire java toString() will be executed without calling, that’s why we are calling this special method.
   * **Example:** System.out.println(new Object().toString());
2. **equals(Object obj): boolean**
   * Compares this object with the specified object for equality.
   * **Example:** String s1 = "hello"; String s2 = "hello"; System.out.println(s1.equals(s2));
3. **hashCode(): int**
   * Returns a hash code value for the object.
   * **Example:** Integer i = new Integer(10); System.out.println(i.hashCode());
4. **getClass(): Class<?>**
   * Returns the Class object associated with this object.
   * **Example:** String s = "hello"; System.out.println(s.getClass());
5. **clone(): Object**
   * Creates and returns a copy of this object.
   * **Example:** Object obj = new Object(); Object cloneObj = obj.clone();
6. **wait(): void**
   * Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object if developer for to call notify() then thread will go to dead lock situation.
   * **Example:** Used in inter-thread communication.
7. **wait(long timeout): void**
   * Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or a specified timeout occurs.
   * **Example:** Used in inter-thread communication with a timeout.
8. **wait(long timeout, int nanos): void**
   * Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or a specified timeout occurs, with nanosecond precision.
   * **Example:** Used in inter-thread communication with a precise timeout.
9. **notify(): void**
   * Wakes up a single thread that is waiting on this object's monitor.
   * **Example:** Used in inter-thread communication to signal a waiting thread.
10. **notifyAll(): void**

* Wakes up all threads that are waiting on this object's monitor.
* **Example:** Used in inter-thread communication to signal all waiting threads.

1. **finalize(): void**

* Called by the garbage collector on an object before it is garbage collected.
* **Example:** Can be overridden to perform cleanup tasks, but it's generally discouraged.

**String :**

String is an Immutable object in java, as well as it is a predefined class. in simple term string is a group of character.

We can create string object by 2 way :-

1. By new key word approach.
2. By literals approach.

**New Keyword:**

The new keyword is used to create a new object in memory every time because of new keyword.

**Object Reference Variable:** The str variable stores the memory

address of the String object.

**By literal approach :**

We use “” double code for creating string object. It will not create new object every time if object is already exist then.

**Property:**

**String Elements:** The String object contains a sequence of characters.

Each character has an index, starting from 0.

**Default Value:** default value of string is null.

**Length:** The length() method can be used to get the length of a String object.

**Immutable:** Once created, their object cannot be changed. if we try to

change then it will change with newly created object.

**String Pool:** Java uses a String pool to optimize memory usage for strings.

When you create a String object, the JVM checks the String pool to see

if an identical string already exists. If it does, the existing object is

returned instead of creating a new one.

**Methods in string :**

* **length()**: i**nteger**

Returns the length of the string.

* **charAt(index): character**

Returns the character at the specified index.

* **toCharArray() : char[ ]**

method is a built-in method in the String class that converts a given string into a character array.

* **concat(str)**: **String**

add substring to the end of this string.

* **indexOf(str): integer**

Returnsthe index of the first occurrence of the specified substring.

* **lastIndexOf(str): integer**

Returns the index of the last occurrence of the specified substring.

* **substring(beginIndex): string**

Returns a substring starting from the specified index to the last.

* **substring(beginIndex, endIndex)**: string

Returns a substring starting from the specified begin Index and ending Index between.

* **toLowerCase(): string**

Converts all characters to lowercase.

* **toUpperCase(): string**

Converts all characters to uppercase.

* **trim(): string**

Removes white space from first and last.

* **startsWith(prefix): Boolean**

Checks if the string starts with the specified prefix.

* **endsWith(suffix): Boolean**

Checks if the string ends with the specified suffix.

* **replace(oldChar, newChar): string**

Replaces all occurrences of the old character with the new character.

* **split(delimiter): string**

Splits the string into an array of substrings based on the delimiter.

* **equals(str): Boolean**

Compares two strings for equality.

* **equalsIgnoreCase(str): Boolean**

Compares two strings for equality, ignoring case.

* **compareTo() or compareToIgnoreCase() : int**

The compareTo() method in Java is used to compare two strings lexicographically.

**#difference b/w String , string builder and string buffer?**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| Mutability | Immutable | Mutable | Mutable |
| Thread Safety | Not thread-safe | Not thread-safe | Thread-safe |
| Performance | Less efficient for frequent modifications | More efficient for frequent modifications | Less efficient than StringBuilder for single-threaded environments |
| Use Cases | When you need a constant string | When you need to modify strings frequently in a single-threaded environment | When you need to modify strings frequently in a multi-threaded environment |

#common methods in wrapper class?

|  |  |  |
| --- | --- | --- |
| Primitive Data Type | Wrapper Class | Common Methods |
| byte | Byte | byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue(), toString(), valueOf() |
| short | Short | byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue(), toString(), valueOf() |
| int | Integer | byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue(), toString(), valueOf(), parseInt(), toHexString(), toOctalString(), toBinaryString() |
| long | Long | byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue(), toString(), valueOf(), parseLong() |
| float | Float | byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue(), toString(), valueOf(), parseFloat() |
| double | Double | byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue(), toString(), valueOf(), parseDouble() |
| boolean | Boolean | booleanValue(), toString(), valueOf(), parseBoolean() |
| char | Character | charValue(), toString(), valueOf(), isDigit(), isLetter(), isLetterOrDigit(), isLowerCase(), isUpperCase(), toLowerCase(), toUpperCase() |

**Math Class:**

It provides a set of mathematical functions that can be used to perform various mathematical operations. It's a static class, meaning you don't need to create an instance of it to use its methods.

**Methods in math class:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Method | Allow data type | Description |
|  | abs(int a) | Long, float, double, int | Returns the absolute value of an integer. |
|  | ceil(double a) | double | Returns the smallest integer that is greater than or equal to the argument. |
|  | floor(double a) | Double, float | Returns the largest integer that is less than or equal to the argument. |
|  | round(double a) | Double, float | Returns the closest long to the argument. |
|  | min(int a, int b) | Int, long, float, double | Returns the smaller of two integers. |
|  | max(int a, int b) | Int, long, float, double | Returns the larger of two integers. |
|  | sqrt(double a) | Int, float, double | Returns the square root of a double. |
|  | pow(double a, double b) |  | Returns the value of a raised to the power of b. |
|  | log(double a) |  | Returns the natural logarithm of a double. |
|  | log10(double a) |  | Returns the base 10 logarithm of a double. |
|  | sin(double a) |  | Returns the sine of an angle in radians. |
|  | cos(double a) |  | Returns the cosine of an angle in radians. |
|  | tan(double a) |  | Returns the tangent of an angle in radians. |
|  | asin(double a) |  | Returns the arcsine of a value in radians. |
|  | acos(double a) |  | Returns the arccosine of a value in radians. |
|  | atan(double a) |  | Returns the arctangent of a value in radians. |
|  | toRadians(double angdeg) |  | Converts degrees to radians. |
|  | toDegrees(double angrad) |  | Converts radians to degrees. |
|  | random() |  | Returns a random number between 0.0 (inclusive) and 1.0 (exclusive). |

#Buble sort :

public class BubbleSort {

public static void bubbleSort(int[] arr) {

int n = arr.length;

boolean swapped;

for (int i = 0; i < n - 1; i++) {

swapped = false;

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap arr[j] and arr[j+1]

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = true;

}

}

// If no two elements were swapped in inner loop, then break

if (!swapped)

break;

}

}

public static void main(String[] args) {

int[] arr = {64, 34, 25, 12, 22, 11, 90};

bubbleSort(arr);

System.out.println("Sorted array");

for (int i = 0; i < arr.length; ++i)

System.out.print(arr[i] + " ");

System.out.println();

}

}

#Diff b/w OOPs, POPs and SOA ?

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | OOP | PoPs | SOA |
| Focus | Objects and their interactions | Procedures and functions | Services |
| Granularity | Classes and objects | Functions | Services |
| Coupling | Loose coupling between objects | Tight coupling between functions | Loose coupling between services |
| Reusability | High reusability through inheritance and polymorphism | Lower reusability | High reusability through service interfaces |
| Scalability | Can be scalable but requires careful design | Less scalable | Highly scalable due to service-based architecture |
| Example | Java, C++ | C, Pascal | Microservices architecture |

**Class**

* A user-defined data type.
* Creates a new data type when defined.
* Acts as a blueprint for objects.
* A single class can create multiple objects.

**Interface**

* A user-defined data type.
* Creates a new data type when defined.
* Acts as a blueprint for objects.
* interface cannot create objects.

**Object**

* A real-world entity.
* Composed of state (data) and behavior (methods).
* Created using the new keyword.

**#Diff b/w method overloading and overriding?**

|  |  |  |
| --- | --- | --- |
| Feature | Method Overloading | Method Overriding |
| Occurrence | Within the same class | In inheritance is must |
| definition | Multiple method with same name but diff data type of arg. | Multiple method with same method signature but diff data type of argument. |
| Parameter List | Different parameter lists (number, type, or both) | Same parameter list |
| Return Type | Can have different return types | Must have the same return type |
| Access Modifiers | Can have different access modifiers | Overriding method's access modifier cannot be more restrictive |
| Polymorphism | Compile-time polymorphism | Runtime polymorphism |
| With method | Static and non-static | Non-static |

**Exa : method overloading**

class Calculator {

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

}

**Exam : method overriding**

class Animal {

public void sound() {

System.out.println("Animal sound");

}

}

class Dog extends Animal {

@Override

public void sound() {

System.out.println("Woof!");

}

}

**Compile-time Binding / Early Binding / Static Binding**

* **Static methods** are bound during compile time.
* This is because there's only one copy of a static method, regardless of the number of objects created.
* The compiler directly links the method call to the specific method implementation at compile time.

**Runtime Binding / Late Binding / Dynamic Binding**

* **Non-static methods** (instance methods) are bound during runtime.
* This is due to the multiple copies of instance methods, one for each object.
* At runtime, the JVM determines the appropriate method implementation based on the object's actual type.

|  |  |  |
| --- | --- | --- |
| Feature | Method Hiding | Method Overriding |
| Inheritance | Not necessary | Inheritance is required |
| Method Type | Static methods | Non-static methods |
| Access Modifiers | Can have different access modifiers | Overriding method's access modifier cannot be more restrictive |
| Behaviour | Hides the superclass method | Overrides the superclass method |

**\*\*\*Inheritance(imp)\*\*\***

* It is a concept which is use to make sub class based on the parent class or existing class. by using extends keyword.
* Inheritance give each property of supper class to the sub class (for non-static)
* Static member can not inherit it can share to each class.
* If there is inheritance b/w class and having constructor then constructor chaining is mandatory by using supper() statement.
* Default constructor do not need to chain it is doing by jvm implicitly.

Exa:

class Animal {

public void eat() {

System.out.println("Animal is eating");

}

}

class Dog extends Animal {

public void bark() {

System.out.println("Dog is barking");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // Inherited from Animal

dog.bark();

}

}

**Types of inheritance:**

1. Single inheritance
2. Multi level inheritance
3. Hierarchical inheritance
4. Multiple inheritance
5. **Single inheritance**

1 class exted the property of another class is

Animal

Dog

Called single level of inheritance.

class Animal {

}

class Dog extends Animal {

}

Exa :

Single Inheritance:

class Vehicle {

protected String brand;

protected int year;

public Vehicle(String brand, int year) {

this.brand = brand;

this.year = year;

}

}

class Car extends Vehicle {

private String model;

public Car(String brand, int year, String model) {

super(brand, year); // Call the parent constructor

this.model = model;

}

}

1. **Multilevel inheritance**

One class inherit the property of parent class and that sub class again inherit 1 sub class is classed multilevel inheritance.

class Animal {

}

class Dog extends Animal {

}

class Puppy extends Dog {

}

Exa :

Animal

Dog

Puppy

interface Flyable {

void fly();

}

interface Swimmable {

void swim();

}

class Duck implements Flyable, Swimmable {

public void fly() {

System.out.println("Duck is flying");

}

public void swim() {

System.out.println("Duck is swimming");

}

}

1. **Hierarchical inheritance**

One super class having multiple sub class is called hierarchical inheritance.

class Animal {

}

class Dog extends Animal {

}

class Cat extends Animal {

Animal

}

Exa :

protected String brand;

protected int year;

public Vehicle(String brand, int year) {

this.brand = brand;

this.year = year;

}

public void start() {

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

}

}

class Car extends Vehicle {

private String model;

public Car(String brand, int year, String model) {

super(brand, year);

this.model = model;

}

public void accelerate() {

System.out.println("Car is accelerating");

}

}

class Bike extends Vehicle {

public void brake() {

System.out.println("Bike is braking");

}

}

**4) Multiple inheritance\*\*\***

Multiple inheritance is not possible in java class.Because of Dimond and ambiguity problem.

\*Single subclass havening multiple supper class is called Multiple inheritance

A screenshot of a computer

Description automatically generated

**Inheritance By Using Interface :**

1. **Single level inheritance :**

interface Animal {

void eat();

void sleep();

}

class Dog implements Animal {

public void eat() {

System.out.println("Dog is eating");

}

public void sleep() {

System.out.println("Dog

is sleeping");

}

}

1. Multiple-Level Inheritance

interface Pet {

void pet();

}

interface Animal extends Pet {

void eat();

void sleep();

}

class Dog implements Animal {

public void eat() {

System.out.println("Dog is eating");

}

public void sleep() {

System.out.println("Dog is sleeping");

}

public void pet() {

System.out.println("Petting the dog");

}

}

1. **Multiple Inheritance**

interface Flyable {

void fly();

}

interface A{

}

interface Swimmable extends A{

void swim();

}

class Duck implements Flyable, Swimmable {

public void fly() {

System.out.println("Duck is flying");

}

public void swim() {

System.out.println("Duck is swimming");

}

}

1. **Hierarchical Inheritance**

interface Vehicle {

void move();

}

interface Car extends Vehicle {

void drive();

}

interface Bike extends Vehicle {

void ride();

}

class Sedan implements Car {

public void move() {

System.out.println("Sedan is moving");

}

public void drive() {

System.out.println("Driving a sedan");

}

}

class Motorcycle implements Bike {

public void move() {

System.out.println("Motorcycle is moving");

}

public void ride() {

System.out.println("Riding a motorcycle");

}

}

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

Note:

* Class extends class
* Class extends class and implements interface
* Class implements multiple interfaces
* Interface extends interface
* Interface extends multiple interfaces

**\*\*\*Encapsulation\*\*\***

* binding data members (variables and methods) with into a single unit (class) is called encapsulation.
* Steps of Encapsulation:
  1. Declare variables as private.
  2. Provide access to the variables through public methods.
  3. Provide validation conditions inside the methods.

class EncapsulationExample {

private int age; // Private variable

public int getAge() { // Public getter method

return age;

}

public void setAge(int age) { // Public setter method

if (age >= 0) {

this.age = age;

} else {

System.out.println("Invalid age");

}

}

}

public class Main {

public static void main(String[] args) {

EncapsulationExample person = new EncapsulationExample();

person.setAge(25); // Set the age through the setter

int age = person.getAge(); // Get the age through the getter

System.out.println("Age: " + age);

}

}

**# Abstraction(imp) \*\*\***

* In simple words, abstraction means hiding unnecessary detail and providing access only to necessary detail.
* In technical terms, abstraction means hiding method implementation detail and providing access only to method signature.

**Abstraction is achieved in 2 ways: By using abstract class and by using interface.**

**Steps to design Abstraction:**

**Step 1:** Create an interface / abstract class.

**Step 2:** Create a method in interface / abstract class.

**Step 3:** Create implementation class / sub class.

**Step 4:** Implement the method / Override the method.

**Step 5:** Create a helper class.

**Step 6:** Create helper methods.

**Step 7:** Helper method will create object of implementation class / sub class.

**Step 8:** Implementation class object is upcasted to interface / superclass.

**Step 9:** Upcasted object is returned.

**Step 10:** Call the helper method.

* If the helper method is static, call the helper method by using helper class name.
* If the helper method is non-static, create helper class object and call the helper method by using helper class object reference variable.

**Step 11:** Helper method will return implementation class object in upcasted manner.

**Step 12:** Receive the object by using interface datatype followed by object reference

variable.

**Step 13:** Use object reference variable to call interface methods.

**Exm :1**

interface Shape {

void draw();

}

class Circle implements Shape {

@Override

public void draw() {

System.out.println("Drawing a circle");

}

}

class HelperClass {

public static Shape getShapeObject() {

Shape shape=new Circle();

return shape;

}

}

public class Main {

public static void main(String[] args) {

Shape shape = HelperClass.getShapeObject();

shape.draw(); // Output: Drawing a circle

}

}

**#diff b/w abstract and abstraction?**

|  |  |  |
| --- | --- | --- |
| Feature | Abstract Class | Abstraction |
| Definition | It means incomplete class which having at list 1 incomplete method. | Hiding unnecessary detail, in java hiding the implementation class detail give access to the method signature. |
| Methods | Can have both abstract and concrete methods | Can be implemented in abstract classes or interfaces |
| Inheritance | Single inheritance | Multiple inheritance through interfaces |
| Variables | Can have variables of all types | Not directly related to variables |
| Purpose | To provide a partial implementation and define a base class for subclasses | To simplify complex systems and reduce code redundancy |

**# Polymorphism**

* Polymorphism means single entity having multiple forms.
* Polymorphism classified into two types:
  1. **Compile-time polymorphism**
  2. **Run-time polymorphism**
* **Compile-time polymorphism\*\*\***
* Call to overloaded method is decided during compilation time based on the argument list. This is called compile-time polymorphism.
* In order to achieve compile-time polymorphism, we need to use:
  1. Static methods
  2. Method overloading
* **Run-time polymorphism\*\*\***
* Call to overridden method is decided during runtime based on the object creation. This is called runtime polymorphism.
* To achieve runtime polymorphism, we need to use:
  1. Interface/Superclass/abstract class
  2. Implementation class/subclass
  3. Method overriding.
  4. Generalization
  5. Upcasting

**Exa : Compile time polymorphism\*\*\***

class Calculator {

public static int add(int a, int b) {

return a + b;

}

public static double add(double a, double b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

int sum1 = Calculator.add(2, 3);

double sum2 = Calculator.add(2.5, 3.5);

}

}

**Exa : Run time polymorphism**

class Animal {

public void makeSound() {

System.out.println("Generic animal sound");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Woof!");

}

}

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("Meow!");

}

}

public class Main {

public static void main(String[] args) {

Animal animal1 = new Dog();

Animal animal2 = new Cat();

animal1.makeSound(); // Output: Woof!

animal2.makeSound(); // Output: Meow!

}

}

\*\*\*Instanceof key word \*\*\*

It is used to check if an object is belong to particular class / interface or not, ensuring type safety during down casting.

Syntex : boolean result = object instanceof ClassName;

**Exa :**

class Animal {

public void makeSound() {

System.out.println("Generic animal sound");

}

}

class Dog extends Animal {

public void bark() {

System.out.println("Woof!");

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Dog();

if (animal instanceof Dog) {

Dog dog = (Dog) animal; 1

dog.bark(); // This will print "Woof!"

} else {

System.out.println("Animal is not a Dog");

}

}

}

public class NeonNumber {

public static void main(String[] args) {

int num = 9, square = num \* num, sum = 0;

// Calculate the sum of the digits of the square

while (square > 0) {

sum += square % 10;

square /= 10;

}

if (sum == num)

System.out.println(num + " is a Neon Number.");

else

System.out.println(num + " is not a Neon Number.");

}

**\*\*\*Constructor (imp)\*\*\***

* A constructor is a special method that is executed during object creation time.
* The constructor's name must be the same as the class name.
* Constructors are by default non-static and public. Developers should not use the static keyword with constructors.

Syntax of a constructor:

accessSpecifier ConstructorName(arguments/non- arguments) {

// Constructor body

}

* If a developer doesn't create any constructor, the compiler will create a default constructor. a constructor without arguments and with an empty body is called a default constructor.
* Constructors are classified into two types:
* Constructor with arguments
* Constructor without arguments
* Constructors are used for performing critical operations.
* One of the common critical operations in every program is the initialization of non-static variables.
* **Creating multiple constructors with different argument data types is called constructor overloading.**
* **Calling one constructor from another constructor is called constructor chaining.**
* **A constructor cannot call another constructor by using its name.**
* **To call another constructor, we must use the** this() **statement with in the same class.**
* To call another constructor on other class we use supper() statement.
* this() **statement informs the JVM to call the appropriate constructor based on the argument data type.**
* **Syntax of the** this() **statement:**

this(arg1, arg2, ...);

* **A constructor can chain with only one (single) constructor.**
* **To avoid breaking the above rule, the** this() **statement must be the first statement in the constructor.**

**Exa:1**

class Box {

int width, height, depth;

// Default constructor

Box() {

width = height = depth = 5;

}

// Parameterized constructor

Box(int w, int h, int d) {

width = w;

height = h;

depth = d;

}

}

**Exa:2 Argumenta, non argumenta constructor and constructor overloading.**

class Car {

private String model;

private int year;

// Non-argumented constructor (default constructor)

public Car() {

model = "TATA";

year =2024;

}

// Argumented constructor

public Car(String model, int year) {

this.model = model;

this.year = year;

}

public void displayInfo() {

System.out.println("Model: " + model);

System.out.println("Year: " + year);

}

}

public class Main {

public static void main(String[] args) {

// Creating a car object using the default constructor

Car car1 = new Car();

car1.displayInfo();

// Creating a car object using the parameterized constructor

Car car2 = new Car("Toyota Camry", 2023);

car2.displayInfo();

}

}

**Exa : 3 Constructor chaining by using this()**

class Car {

private String model;

private int year;

// Default constructor

public Car() {

this("Unknown", 0); // Chaining to the parameterized constructor

}

// Parameterized constructor

public Car(String model, int year) {

this.model = model;

this.year = year;

}

}

**Exm : 3 chaining with diff class by using supper key words.**

class Parent {

private String parentName;

public Parent(String parentName) {

this.parentName = parentName;

System.out.println("Parent class constructor called");

}

}

class Child extends Parent {

private String childName;

public Child(String childName) {

super("Parent Name"); // Calling the parent class's constructor

this.childName = childName;

System.out.println("Child class constructor called");

}

}

public class Main {

public static void main(String[] args) {

Child child = new Child("Child Name");

}

}

**Exa : 4 Constructor chaining by using supper() and this()**

class Vehicle {

protected String brand;

protected int year;

public Vehicle(String brand, int year) {

this.brand = brand;

this.year 1 = year;

}

}

class Car extends Vehicle {

private String model;

public Car(String brand, int year, String model) {

super(brand, year); // Call the parent class's constructor

this.model = model;

}

}

public class Main {

public static void main(String[] args) {

Car car = new Car("Toyota", 2023, "Camry");

}

}

|  |  |  |
| --- | --- | --- |
| Feature | Is-A Relationship (Inheritance) | Has-A Relationship (Composition/Aggregation) |
| Keyword | extends | No specific keyword |
| Relationship | Hierarchical | Whole-part |
| Access | Inherited members can be accessed directly | Members of the composed object need to be accessed through its reference |
| Lifetime | Lifetime of the child object is tied to the parent object | Lifetime of the composed object is independent of the container object |

**#Diff. b/w Aggregation and composition ?**

|  |  |  |
| --- | --- | --- |
| Feature | Composition | Aggregation |
| Relationship Type | Strong | Weak |
| Keyword | has-a relationship | has-a relationship, but less tightly coupled |
| Representation | Solid diamond | Hollow diamond arrow |
| Object Creation | Objects are created together and destroyed together. | Objects can exist independently. |
| Example | Car has an Engine | Library has Books |

**#static and non-static block?**

|  |  |  |
| --- | --- | --- |
| Feature | Static Block | Non-Static Block |
| Execution | Once, when the class is loaded | Multiple times, when objects are created |
| Scope | Class level | Object level |
| Purpose | Initialization tasks for the entire class | Initialization tasks for individual objects |

**Exa : static block**

public class MyClass {

static {

System.out.println("Static block executed");

}

public static void main(String[] args) {

System.out.println("Main method executed");

}

}

**Exa : non-static block**

public class MyClass {

{

System.out.println("Non-static block executed");

}

public static void main(String[] args) {

MyClass obj = new MyClass();

}

}

**The final Keyword in Java**

The final keyword in Java is used to declare entities that cannot be modified once they are initialized. It can be applied to variables, methods, and classes.

**1. Final Variables:**

If we declare variable once final we cannot re-initialize.

* **Constant Variable**

final double PI = 3.14159;

* **Class-Level Constants:**

public class MyClass {

public static final int MAX\_VALUE = 100;

}

* **Final Methods:**

If we declare method once final we cannot override that method in the sub class.

* **Preventing Overriding:**

class Parent {

public final void method () {

// code

}

}

class Child extends Parent {

// Cannot override the final method

public void method () {

// code

}

}

* **Final Classes:**

If we declare class once final, we cannot inherit that class.

**Preventing Inheritance:**

final class ImmutableClass {

// code

}

class DerivedClass extends ImmutableClass { // Error: Cannot inherit from a final class

// code

}

**#diff b/w local variable and global variable ?**

|  |  |  |
| --- | --- | --- |
| Feature | Local Variable | Global Variable |
| Scope | Limited to method/block | Accessible throughout the class |
| Lifetime | Exists only during method/block execution | Exists as long as the class is loaded |
| Accessibility | Only accessible within the method/block | Accessible within the class and subclasses |
| Initialization | Must be explicitly initialized | Can be explicitly initialized or assigned a default value |
| Default Value | No default value | 0 for numeric types, false for boolean, null for references |

**#diff b/w static global variable and non-static global variable ?**

|  |  |  |
| --- | --- | --- |
| Feature | Static Global Variable | Non-Static Global Variable |
| Scope | Class-level | Instance-level |
| Lifetime | Class lifetime | Object lifetime |
| Memory Allocation | Once per class | Once per object |
| Sharing | Shared among all instances | Specific to each instance |
| Access | Using class name | Using object reference |

**Marker Interface**

* Also known as tagging interfaces.
* Doesn't declare any methods.
* Used to mark or tag classes that need to undergo specific behaviors or processing.
* **Examples:**
  + java.io.Serializable: Used for serializing objects.
  + java.lang.Cloneable: Used for cloning objects.
  + java.util.RandomAccess: Used for efficient random access to list elements.

**Functional Interface**

* Contains exactly one abstract method.
* Used to take advantage of lambda expressions and method references in Java 8.
* **Example:**

interface Calculator {

int operation(int a, int b);

}

**Default Method Interface**

* Introduced in Java 8.
* Provides default implementations for interface methods.
* Classes implementing the interface can use the default implementation or provide their own.
* **Example:**

interface MyInterface {

default void myMethod() {

System.out.println("Default implementation");

}

}

**Key Points:**

* **Marker Interfaces:** Used for tagging and metadata purposes.
* **Functional Interfaces:** Used for concise and functional programming style.
* **Default Method Interfaces:** Provide flexibility and reduce boilerplate code in interface implementations.

#**Introduced features ?**

|  |  |
| --- | --- |
| JDK Version | Major Features |
| Java 1.1 | Inner classes, JDBC, RMI, JavaBeans |
| Java 1.2 | Collections Framework, Swing, JIT compiler |
| Java 8 (1.8) | Lambda expressions, Stream API, Functional interfaces, Date and Time API, Optional class |
| Java 11 | HTTP Client API, Epsilon Garbage Collector, ZGC: A Scalable Low-Latency Garbage Collector |
| Java 17 | Sealed Classes, Pattern Matching for instanceof and switch, Records, Foreign-Memory Access API (Standard Module) |

**Object Casting :**

**Object Casting** is the process of converting one object to look like a other. In Java, this is possible only when there's a inheritance relationship between the two classes, i.e., one class is a subclass of the other.

**Types of Object Casting:**

1. **Upcasting:**
   * Converting a subclass object to its superclass.
   * This is implicit and happens automatically.
   * Subclass properties are hidden during upcasting.

Animal animal = new Dog(); // Upcasting

1. **Downcasting:**
   * Converting a superclass object to its subclass.
   * This is explicit and requires a cast operator.
   * Down casting is possible only after upcasting and can lead to ClassCastException if the cast is invalid.

Dog dog = (Dog) animal; // Downcasting

**Example:**

class Animal {

public void eat() {

System.out.println("Animal is eating");

}

}

class Dog extends Animal {

public void bark() {

System.out.println("Dog is barking");

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Dog(); // Upcasting

animal.eat(); // Output: Animal is eating

Dog dog = (Dog) animal; // Downcasting

dog.bark(); // Output: Dog is barking

}

}

**What is Collection (Interface)?**

* Collection is a predefined interface in Java.
* It was introduced in version JDK 1.2.
* It is present in the java.util package.
* In Java, the collection framework provides a standard architecture to store groups of objects.

**Note Point:**

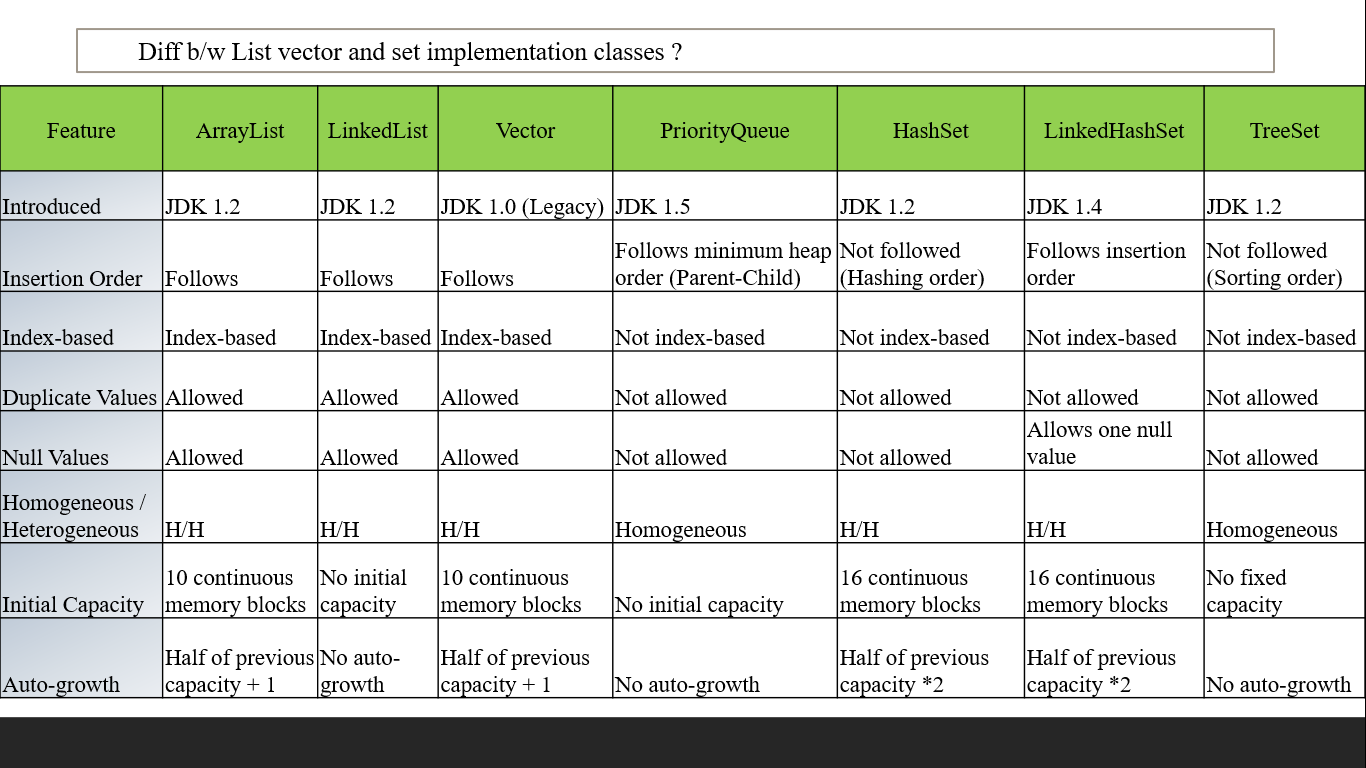
* Collection is nothing but a collection/group of objects.

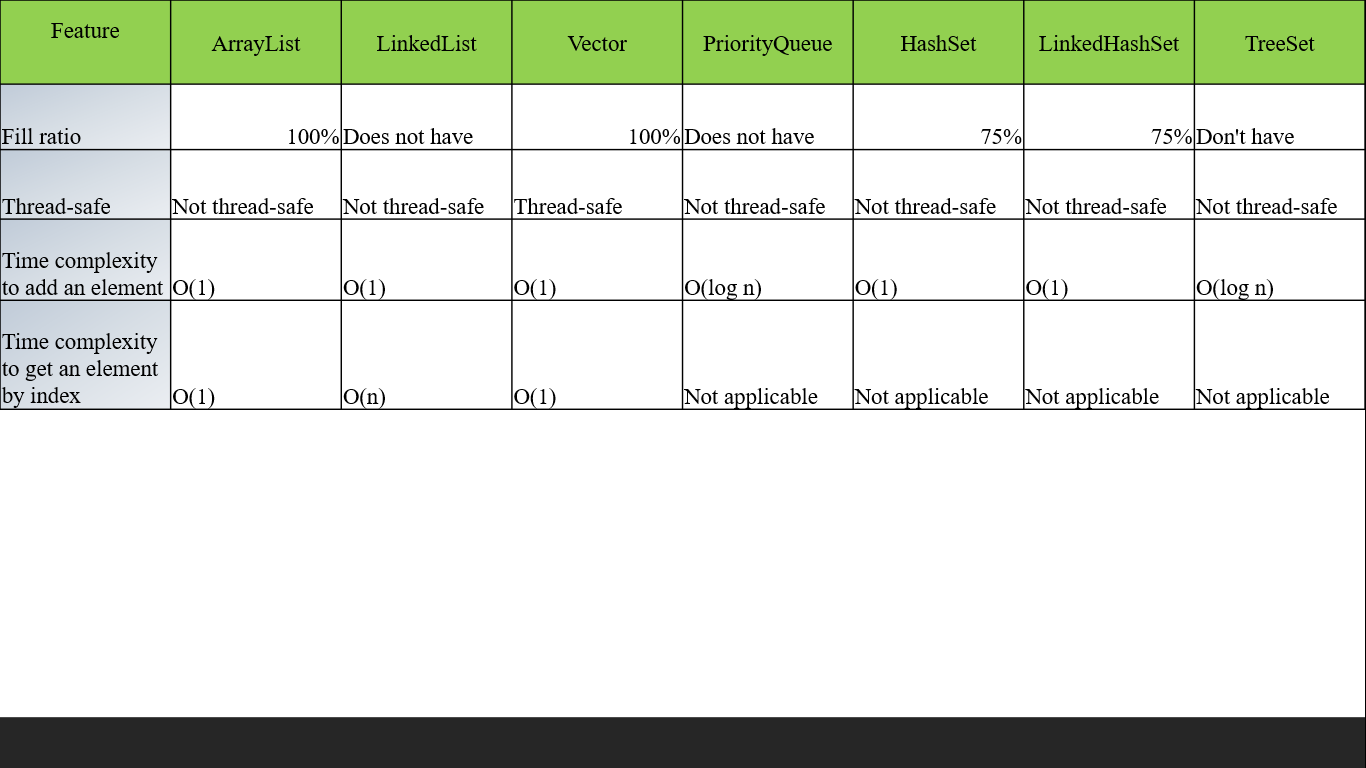
**What is Framework?**

* Framework represents a group of classes and interfaces.

A screenshot of a computer

Description automatically generated





Exa : of ArrayList with all method and cursor(iterator).

import java.util.ArrayList;

import java.util.Iterator;

import java.util.ListIterator;

public class ArrayListExample {

public static void main(String[] args) {

// Create an ArrayList

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

// Add elements to the ArrayList

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

fruits.add("Mango");

// Print the ArrayList

System.out.println("Fruits: " + fruits);

// Get the size of the ArrayList

int size = fruits.size();

System.out.println("Size of the ArrayList: " + size);

// Check if the ArrayList is empty

boolean isEmpty = fruits.isEmpty();

System.out.println("Is the ArrayList empty? " + isEmpty);

// Access elements by index

String firstFruit = fruits.get(0);

System.out.println("First fruit: " + firstFruit);

// Set an element at a specific index

fruits.set(1, "Grapes");

System.out.println("Modified fruits: " + fruits);

// Remove an element by index

fruits.remove(2);

System.out.println("Fruits after removal: " + fruits);

// Remove an element by value

fruits.remove("Banana");

System.out.println("Fruits after removal: " + fruits);

// Check if an element exists

boolean containsMango = fruits.contains("Mango");

System.out.println("Does the ArrayList contain Mango? " + containsMango);

// Clear the ArrayList

fruits.clear();

System.out.println("After clearing: " + fruits);

// Iterate using a for-each loop

for (String fruit : fruits) {

System.out.println(fruit);

}

// Iterate using an Iterator

Iterator<String> iterator = fruits.iterator();

while (iterator.hasNext()) {

String fruit = iterator.next();

System.out.println(fruit);

}

// Iterate using a ListIterator

ListIterator<String> listIterator = fruits.listIterator();

while (listIterator.hasNext()) {

String fruit = listIterator.next();

System.out.println(fruit);

}

// Add elements using ListIterator

listIterator.add("Apple");

listIterator.add("Banana");

System.out.println("Fruits after adding: " + fruits);

}

}

**\*\*\*Cursor\*\*\***

* What is the purpose of cursor?
  + The purpose of cursor is to retrieve the object one by one from the collection.
* What is cursor?
  + Cursor is nothing but traversing.
  + Traversing means cursor should move from one object to another object.
* In Java we have 3 predefined cursors:
  + Iterator
  + ListIterator
  + Enumeration

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | Iterator | ListIterator | Enumeration  (lacy cursor) |
| Introduced in | JDK 1.2 | JDK 1.2 | JDK 1.0 |
| Direction | Forward only | Forward and backward | Forward only |
| Removal | Can remove elements | Can remove, add, and replace elements | Cannot remove elements |
| Methods | hasNext(), next() | hasNext(), next(), hasPrevious(), previous(), nextIndex(), previousIndex(), add(), set(), remove() | hasMoreElements(), nextElement() |

**Map Object**

* In map, we can store values in key and value pair.
* Map is an interface which is having 3 implementation classes.
  1. HashMap
  2. LinkedHashMap
  3. TreeMap

**Map Object** will be containing object in the form of key and value.

* Key should not be duplicated and value can be duplicated.
* Each key and value pair are known as entry object.
* Because of this reason Map Object is also known as collection of entry objects.
* Whenever we want to store a single object, go with collection, whenever we want to store in the form of key and value pair at that time make use of Map Object.

#**diff b/w HashMap, LikedHashMap and TreeMap?**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | HashMap | LinkedHashMap | TreeMap | HashTable |
| Introduced | JDK 1.2 | JDK 1.4 | JDK 1.2 | JDK 1.0 |
| Ordering | No specific order | Insertion order | Sorted order | No specific order |
| Null Keys and Values | Allows one null key, multiple null values | Allows one null key, multiple null values | Does not allow null keys | Does not allow null keys or values |
| Thread Safety | Not thread-safe | Not thread-safe | Not thread-safe | Thread-safe |
| Performance | Generally fastest | Slower than HashMap but faster than TreeMap | Slower than HashMap for insertion and retrieval. | Slower than HashMap due to synchronization |
| Output order | Hashing technique order | Insertion order | Sorting order |  |

**#methodes in Map :**

|  |  |
| --- | --- |
| Method | Description |
| put(K key, V value) | Inserts a key-value pair into the map. |
| get(Object key) | Retrieves the value associated with the specified key. |
| remove(Object key) | Removes the key-value pair associated with the specified key. 1 |
| containsKey(Object key) | Returns true if the map contains the specified key. |
| containsValue(Object value) | Returns true if the map contains the specified value. |
| size() | Returns the number of key-value pairs in the map. |
| isEmpty() | Returns true if the map is empty. |
| keySet() | Returns a Set view of the keys contained in this map. |
| values() | Returns a Collection view of the values contained in this map. 2 |
| entrySet() | Returns a Set view of the key-value pairs contained in this map. |
| clear() | Removes all of the mappings from this map. |

#**differece b/w map and collection?**

|  |  |  |
| --- | --- | --- |
| Feature | Collection | Map |
| Data Structure | Stores individual elements | Stores key-value pairs |
| Duplicate Elements | Can contain duplicates | Does not allow duplicate keys |
| Retrieval | Elements are retrieved by index or iterator | Elements are retrieved by key |

**Wrapper classes in java :**

**Purpose:**

* To convert primitive data types into object-oriented data types.
* To enable primitive data types to be used with collections and other object-oriented features.

**Boxing and Unboxing:**

* **Boxing:** Converting a primitive data type to its corresponding wrapper object.
* **Unboxing:** Converting a wrapper object to its corresponding primitive data type.

**Wrapper Classes for Primitive Data Types:**

|  |  |
| --- | --- |
| Primitive Data Type | Wrapper Class |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double 1 |
| char | Character |
| boolean | Boolean 2 |

|  |  |  |
| --- | --- | --- |
| Feature | Boxing | Unboxing |
| Definition | Converting primitive data to object | Converting object to primitive data |
| Ways | Explicit or Implicit | Explicit or Implicit |
| How may way | By 3 way | By 2 way |
| Explicit Exa | Integer i = new Integer(10) | int x = i.intValue() |
| Implicit Exa | Integer i = 10 | int x = i |
| ValueOf() Exa | Integer i = Integer.valueOf(10) |  |

**Exception Handling :**

**What is an Exception?**

* An unexpected event that occurs during the execution of a program.
* When an exception occurs, the normal flow of the program is disrupted, and the program 1 may terminate abnormally.

**Exception Handling:**

* The process of handling exceptions to prevent program crashes and provide graceful error recovery.
* It involves using try, catch, and finally blocks.

**The try-catch-finally Block:**

* **try block:** Encloses the code that might throw an exception.
* **catch block:** Handles specific types of exceptions. Multiple catch blocks can be used.
* **finally block:** Executes regardless of whether an exception is thrown or not. It's often used for cleanup operations like closing files or releasing resources.

**Example**

import java.io.File;

import java.io.FileNotFoundException;

public class ExceptionHandlingExample {

public static void main(String[] args) {

try {

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

Scanner scanner = new Scanner(file);

// ... read from the file

} catch (FileNotFoundException e) {

System.out.println("File not found: " + e.getMessage());

} finally {

// Close the scanner or other resources

// This block will always execute, even if an exception occurs

}

}

}

**Propagation :**

**Exception Propagation** is the process of passing an exception from one method to its caller until it is handled or reaches the top level of the program.

**Types of Exception Propagation:**

1. **Explicit Propagation:**
   * The method that throws the exception uses the throws keyword to declare the exception.
   * The caller method must either handle the exception using a try-catch block or re-throw it using the throws keyword.
   * This is mandatory for checked exceptions.
2. **Implicit Propagation:**
   * The exception is not explicitly thrown using the throws keyword.
   * The exception propagates to the caller method automatically.
   * This is typically the case for unchecked exceptions.

**Types of Exceptions:**

1. **Checked Exceptions:**
   * Checked by the compiler.
   * Must be handled or propagate using the throws keyword.
   * Examples: IOException, SQLException, FileNoteFoundException etc.
2. **Unchecked Exceptions:**
   * Not checked by the compiler.
   * Inherit from the RuntimeException class.
   * Can be propagated implicitly.
   * Exaples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException, IllegalArgumentException, ArithmeticException, NumberFormatException, StringIndexOutOfBoundsException, ConcurrentModificationException, NoSuchElementException etc.

**Example of Explicit Propagation:**

public class ExceptionPropagationExample {

public static void divide(int a, int b) throws ArithmeticException {

if (b == 0) {

throw new ArithmeticException("Division by zero");

}

int result = a / b;

System.out.println("Result: " + result);

}

public static void main(String[] args) {

try {

divide(10, 0);

} catch (ArithmeticException e) {

System.out.println("Error: " + e.getMessage());

}

}

}

**Example of Implicit Propagation:**

public class ImplicitPropagationExample {

public static void divide(int a, int b) {

int result = a / b; // If b is 0, an ArithmeticException will be thrown implicitly

System.out.println("Result: " + result);

}

**Architecture of exception:**

A screenshot of a computer

Description automatically generated

#**diff b/w java, c, c++, and python?**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | C++ | Java | C | Python |
| Type System | Static typing with support for dynamic typing | Static typing | Static typing | Dynamic typing |
| Syntax | Complex, procedural, and object-oriented | Object-oriented | Complex, procedural | Clear and concise |
| Memory Management | Manual (new/delete, smart pointers) | Automatic (Garbage Collector) | Manual (malloc/free) | Automatic (Reference counting) |
| Platform Dependency | Platform-dependent | Write once, run anywhere (WORA) | Platform-specific | Write once, run anywhere (Interpreted) |
| Concurrency | Supports multithreading (Standard library boost) | Supports multithreading (Thread, Executors) | Requires libraries (e.g., pthreads) | Supports multithreading (Global Interpreter Lock in CPython) |
| Library | Need third-party libraries | Standard library | Limited library | Rich standard library |
| Use | System-level programming, game development, embedded systems | Application development, Android apps, web services | System-level programming, embedded systems | Web development, data analysis, automation, scripting |

#**propertys of java , javascript, html, css and sql?**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | Java | JavaScript | HTML | CSS |
| Father | James Gosling | Brendan Eich | Tim Berners-Lee | Håkon Wium Lie |
| Invention | May 1995 | December 1995 | 1993 | 1996 |
| Company | Sun Microsystems (now Oracle) | Netscape Communications (now Mozilla) | CERN | World Wide Web Consortium |
| First Version | JDK 1.0 (1996) | ECMAScript 1 (1997) | HTML 2.0 (1995) | CSS1 (1996) |
| Latest Version | Java 20 (September 2023) | ECMAScript 2023 | HTML5 | CSS3 |
| Definition | High-level, object-oriented programming language. Used to create applications that interact with the real world. | A scripting language used to make web pages interactive. | The standard markup language for creating web pages and web applications. | A style sheet language used to style and layout web pages. |

|  |  |  |
| --- | --- | --- |
| Feature | System.out | System.err |
| Stream Type | Output stream | Error stream |
| Destination | Console or terminal | Console or terminal |
| Behavior | Prints normal program output | Prints error messages and exceptions |
| Syntax for Print | System.out.print() or System.out.println() | System.err.print() or System.err.println() |
| Color | Normal text color (black) | Red color |

#**diff. b/w singly linked list and doubly linked list?**

|  |  |  |
| --- | --- | --- |
| Feature | Singly Linked List | Doubly Linked List |
| Node Structure | Only next reference | Next and previous references |
| Traversal | Forward only | Forward and backward |
| Memory Usage | Less memory usage | More memory usage |
| Reversing | More complex | Easier |
| Insertion/Deletion | Efficient at the beginning, less efficient at the end | Efficient at both ends |

### JIT Compiler

* **Purpose:** Converts bytecode into machine code.
* **Process:** The interpreter initially executes bytecode line by line. The JIT compiler identifies frequently executed code and optimizes it by compiling it into machine code.
* **Benefit:** Improved performance as machine code is executed directly by the processor.

### Why String is Immutable

* **String class is final:** Cannot be subclassed.
* **String pool:** Strings are stored in a string pool, and multiple references can point to the same string object.
* **Security:** Immutability helps in maintaining the security of sensitive data like passwords, network addresses, etc.
* **Efficiency:** String operations often create new strings, but the underlying character array remains the same, reducing memory overhead.

**In essence:**

* **JIT Compiler:** Optimizes performance by converting bytecode to machine code.
* **String Immutability:** Ensures string consistency and security.

## Analyzing the Lambda Expression Code

**Understanding Lambda Expressions**

Lambda expressions are a concise way to represent anonymous functions. They were introduced in Java 8 to simplify functional programming paradigms.

**Breakdown of the Code:**

public interface Addition {

int add(int a, int b);

}

public class Main {

public static void main(String[] args) {

Addition add = (a, b) -> a + b;

int x = add.add(10, 20);

System.out.println(x);

}

}

**servlet class**

1. create a normal class in the project.
2. make a class to behave as implementation class (or) subclass for predefined servlet classes and interface.
3. All predefined classes are present in javax.servlet package
4. javax.servlet package is present in a jar file called servlet-api.jar
5. servlet-api.jar file is present inside Tomcat server

Hierarchy of servlet

**A screenshot of a computer

Description automatically generated**

**Servlet interface**

To create a servlet class by implementing the Servlet interface, you need to override all of its methods.

The Servlet interface defines five methods that must be implemented by any class that implements it:

1. **service():** This method is called by the servlet container to handle incoming requests. It's typically overridden to process requests and generate responses.
2. **destroy():** This method is called by the servlet container when the servlet is being unloaded. It's used to release resources and perform any necessary clean-up.
3. **getServletInfo():** This method returns information about the servlet, such as its author, version, and description.
4. **init():** This method is called by the servlet container when the servlet is first loaded. It's used to initialize the servlet and any resources it needs.
5. **getServletConfig():** This method returns a ServletConfig object, which provides information about the servlet's configuration.

**Exa :**

public class MyServlet implements Servlet {

// Implementation of the five methods

public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {

// Handle the request and generate the response

}

public void destroy() {

// Clean up resources

}

public String getServletInfo() {

return "My Servlet";

}

public void init(ServletConfig config) throws ServletException {

// Initialize the servlet

}

public ServletConfig getServletConfig() {

// Return the servlet configuration

}

}

**GenericServlet**

To create a Servlet class by using the GenericServlet class, you need to override only one method.

The GenericServlet class is an abstract class that provides a basic implementation of the Servlet interface. It has one abstract method, service(), which you must override to handle requests.

Exa :

public class MyServlet extends GenericServlet {

public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {

// Implement the logic to handle the request and generate the response

}

}

**HttpServlet :**

If we want to create a Servlet class by using HttpServlet class we no need to complete any method.

HttpServlet class is an abstract class which is present with all complete methods

public class Myservlet extends HttpServlet

{

// no need to complete any method.

}

**Data Base Connection (JDBC):**

* **Basic Requirements for JDBC Connection:**
  1. **Java Application:** The application that needs to interact with the database.
  2. **JDBC API:** A set of interfaces and classes provided by Java for database connectivity.
  3. **JDBC Driver:** A specific driver for the database system being used (e.g., MySQL, Oracle, SQL Server).
  4. **Database Application:** The database system itself (e.g., MySQL, Oracle, SQL Server).

**Connection Interface:**

* **Purpose:** Provides methods for establishing, managing, and closing a connection to a database.

### DriverManager Class

* **Purpose:** Manages JDBC drivers and provides methods to establish database connections.
* **getConnection() Method:**
  + Takes a database URL as input.
  + Loads the appropriate JDBC driver.
  + Establishes a connection to the database.
  + Returns a Connection object.

### Statement Interface

* **Purpose:** Used to execute static SQL statements.
* **Methods:**
  + executeQuery(): Executes a SELECT query and returns a ResultSet object.
  + executeUpdate(): Executes an INSERT, UPDATE, or DELETE query and returns the number of rows affected.

### PreparedStatement Interface

* **Purpose:** Used to execute parameterized SQL statements, preventing SQL injection attacks.
* **Methods:**
  + setString(): Sets a String value for a parameter.
  + setInt(): Sets an integer value for a parameter.
  + setDouble(): Sets a double value for a parameter. // ... and other setXXX() methods for different data types
  + executeUpdate(): Executes an INSERT, UPDATE, or DELETE query.
  + executeQuery(): Executes a SELECT query and returns a ResultSet object.

### ResultSet Interface

* **Purpose:** Represents a database result set, providing access to the data retrieved from a query.
* **Methods:**
  + next(): Moves the cursor to the next row.
  + first(): Moves the cursor to the first row.
  + last(): Moves the cursor to the last row.
  + absolute(int row): Moves the cursor to the specified row number.
  + relative(int rows): Moves the cursor relative to the current position.
  + getXXX(): Methods to retrieve data from the current row (e.g., getInt(), getString(), getDate()).

**HttpSession Interface :**

* **Purpose:** Represents a session between a client and a server.
* **Methods:**
  + setAttribute(String name, Object value): Stores an attribute in the session.
  + getAttribute(String name): Retrieves an attribute from the session.
  + removeAttribute(String name): Removes an attribute from the session.
  + getMaxInactiveInterval(): Returns the maximum time in seconds that the session can be inactive before it is invalidated.
  + setMaxInactiveInterval(int interval): Sets the maximum time in seconds that the session can be inactive.

### PrintWriter Class

* **Purpose:** Used to write text to a character-output stream.
* **Methods:**
  + print(): Prints a string representation of an object.
  + println(): Prints a string representation of an object followed by a newline character.
  + write(int c): Writes a single character.
  + write(char[] buf): Writes an array of characters.
  + write(String s): Writes a string.

### HttpServletRequest and HttpServletResponse

* **Purpose:** Represents an HTTP request and response, respectively.
* **Methods:**
  + **HttpServletRequest:**
    - getParameter(): Retrieves a parameter value.
    - getParameterValues(): Retrieves an array of parameter values.
    - getSession(): Gets the HTTP session associated with the request.
    - getRequestDispatcher(): Gets a RequestDispatcher object to forward or include another servlet or JSP.
  + **HttpServletResponse:**
    - setContentType(): Sets the content type of the response.
    - getWriter(): Gets a PrintWriter object to write the response.
    - sendRedirect(): Redirects the client to another URL.

### RequestDispatcher Interface

* **Purpose:** Used to forward a request to another resource within the same web application.
* **Methods:**
  + forward(): Forwards the request to another resource within the same application.
  + include(): Includes the content of another resource into the current response.

### Servlet Chaining

**Concept:**

* Involves forwarding a request from one servlet to another within the same web application.
* This allows for modularization and separation of concerns in web applications.

**Implementation:**

1. **Using RequestDispatcher:**
   * Obtain a RequestDispatcher object from the HttpServletRequest object.
   * Use the forward() method to forward the request to the target servlet.

RequestDispatcher dispatcher = request.getRequestDispatcher("/nextServlet");

dispatcher.forward(request, response);

1. **Using @WebServlet Annotation:**
   * Configure the URL pattern for the servlet using the @WebServlet annotation.
   * The servlet container will automatically forward the request to the appropriate servlet based on the URL pattern.

@WebServlet("/nextServlet")

public class NextServlet extends HttpServlet {

// ...

}

Exa:

// FirstServlet.java

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

public class FirstServlet extends HttpServlet {

@Override

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

response.getWriter().println("This is the first servlet.");

request.getRequestDispatcher("/SecondServlet").forward(request, response);

}

}

// SecondServlet.java

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

public class SecondServlet extends HttpServlet {

@Override

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

response.getWriter().println("This is the second servlet.");

}

}

Exa :2

import java.sql.\*;

public class JDBCExample {

public static void main(String[] args) {

try {

// 1. Load the JDBC driver

Class.forName("com.mysql.cj.jdbc.Driver");

// 2. Establish the connection

Connection connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/mydatabase", "username", "password");

// 3. Create a Statement object

Statement statement = connection.createStatement();

// 4. Execute a query

String query = "SELECT \* FROM users";

ResultSet resultSet = statement.executeQuery(query);

// 5. Process the results

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

int age = resultSet.getInt("age");

System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age);

}

// 6. Close resources

resultSet.close();

statement.close();

connection.close();

} catch (ClassNotFoundException | SQLException e) {

e.printStackTrace();

}

}

}

Spring:

**Bean Definition:**

XML

<bean id="myBean" class="com.example.MyBean">

<property name="property1" value="value1" />

<property name="property2" ref="anotherBean" />

<property name="property3">

<value>value3</value>

</property>

<constructor-arg index="0" value="arg1" />

<constructor-arg index="1" ref="anotherBean" />

</bean>

**Key Points:**

* **id attribute:** Unique identifier for the bean.
* **class attribute:** Fully qualified class name of the bean.
* **property element:** Sets properties of the bean using setter injection.
* **constructor-arg element:** Sets constructor arguments using constructor

**Component Scanning:**

<context:component-scan base-package="com.example" />

* Scans the specified package for components annotated with @Component, @Service, @Repository, or @Controller.
* Automatically registers these components as beans in the Spring container.

**Property Placeholder:**

<context:property-placeholder location="classpath:myApp.properties" />

* Loads property values from an external property file.
* Can be used to externalize configuration properties.

Spring jdbc : in spring jdbc don’t having inbuild method to do operation like updte delete and retrive.

Spring JPA : spring jpa having inbuild methods for crud operations .

## JPA Repository Interface

**JPA Repository** is a powerful interface provided by Spring Data JPA that simplifies database operations in Spring applications. It extends the JpaRepository interface from Spring Data Commons, providing a set of predefined methods for common CRUD operations.

|  |  |
| --- | --- |
| Method | Description |
| T save(T entity) | Saves the given entity. If the entity already exists, it updates it. |
| List<T> findAll() | Retrieves all entities. |
| Optional<T> findById(ID id) | Retrieves an entity by its ID. |
| long count() | Counts the number of entities. |
| void delete(T entity) | Deletes the given entity. |
| void deleteById(ID id) | Deletes the entity with the given ID. |
| void deleteAll() | Deletes all entities. |
| boolean existsById(ID id) | Checks if an entity with the given ID exists. |

**Custom Queries:**

You can create custom query methods by using Spring Data JPA's query methods naming conventions or by using the @Query annotation.

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByFirstName(String firstName);

List<User> findByFirstNameAndLastName(String firstName, String lastName);

@Query("SELECT u FROM User u WHERE u.age > ?1")

List<User> findUsersOlderThan(int age);

}