Basic Programs

1) public static void main(String[] args)

Public : It is an Access modifier, which specifies from where and who can access the method. Making the main() method public makes it globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class. If the main method is not public, its access is restricted.

Static: It is a keyword that is when associated with a method, making it a class-related method. The main() method is static so that JVM can invoke it without instantiating the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the main() method by the JVM.If you try to run Java code where main is not static, you will get an error.

Void: It is a keyword and is used to specify that a method doesn't return As the main() method doesn't return anything, its return type is void. As soon as the main() method terminates, the Java program terminates too. Hence, it doesn't make any sense to return from the main() method as JVM can't do anything with its return value .If the main method is not void, we will get an error.

main() : The Java compiler or JVM looks for the main method when it starts executing a Java program. The signature of the main method needs to be in a specific way for the JVM to recognize that method as its entry point. If we change the signature of the method, the program compiles but does not
Execute.

String[] args : is a parameter that accepts String type arguments. It allows us to pass arguments through terminal and it stores these arguments in an array
of strings. We can say that String[] args is a command line argument.

```
package com.basic;
public class Psvm {
	public static void main(String[] args) {
		System.out.println("Java method");
	}
}
```

Output

Java method

We can also change the order of the public , static . main should be always after void only.

Valid Scenarios :

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

public static void main(String... args)

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

public static void main(String []arguments) - (we can rename args)
public static void main(String[] data)

InValid Scenarios :

public void static main(String[] args) -void position. It should be before main only.

static void public main(String[] args) -void position. It should be before main only.

public static void main(int[] args) - The main method must take a single parameter of type String[].

public static main(String[] args) - as void is missing.
public static void mainMethod(String[] args) - as renamed main method name

2) We can Overload the main method.

Overloading the main() method is possible in Java, meaning we can create any number of main() methods in a program.

To overload the main() method in Java, we need to create the main() method with different parameters .

Note: Overriding occurs when a subclass provides its implementation of a method from its superclass. However, the main method is static, and static methods cannot be overridden.

```
package com.basic;
public class OverloadMainMethod {
	public static void main(String[] args) {
		OverloadMainMethod s = new OverloadMainMethod();
		s.main("raju");
		s.main("raju", "rani");
		s.main(10);
		s.main(2000.0);
	}
// by changing the parameter type
	public static void main(String str) {
		System.out.println("name:" + str);
	}
	public static void main(String str, String str1) {
		System.out.println("fullname:" + str + " " + str1);
// by changing the  data  type
	}
	public static void main(int id) {
		System.out.println("id:" + id);
	}
	public static void main(double salary) {
		System.out.println("salary:" + salary);
	}
}
```

Output
name:raju
fullname:raju rani
id:10
salary:2000.0

3) String[]: This is an array of String objects

In the context of String[] args in the main method:
String[]: This is an array of String objects. It is the type of the parameter the JVM passes when invoking the main method.

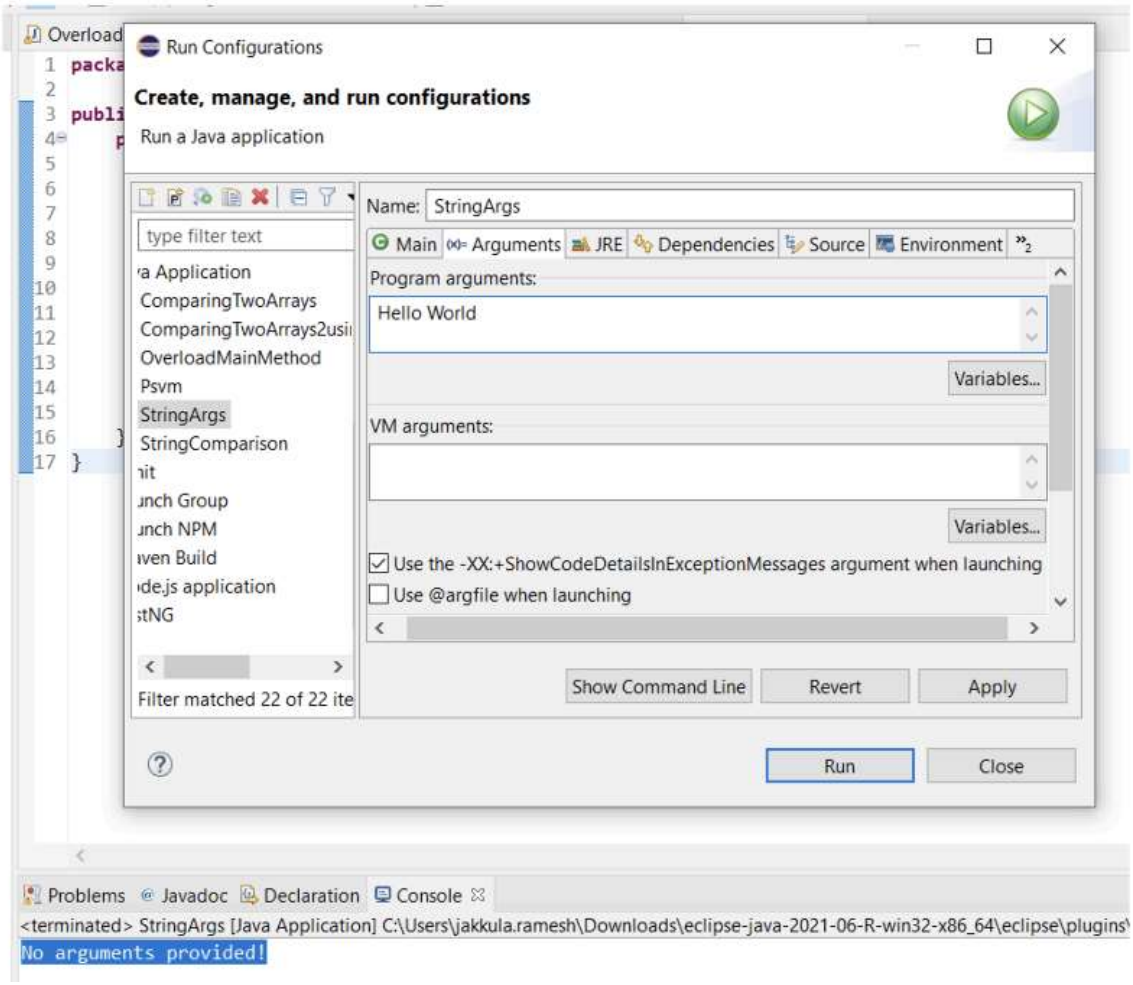
args: This is the name of the parameter, which you can rename to any valid identifier as explained earlier.

In Eclipse: Right-click your program file and select Run As > Run Configurations. In the Arguments tab, enter Hello World in the Program arguments field. Save and run the program.

```
package programms;
public class StringArgs {
	public static void main(String[] args) {
		if (args.length == 0) {
			System.out.println("No arguments provided!");
		} else {
			for (int i = 0; i < args.length; i++) {
				System.out.println("Argument " + i + ": " + args[i]);
			}
		}
	}
}
```

Output

No arguments provided!



Output be like:

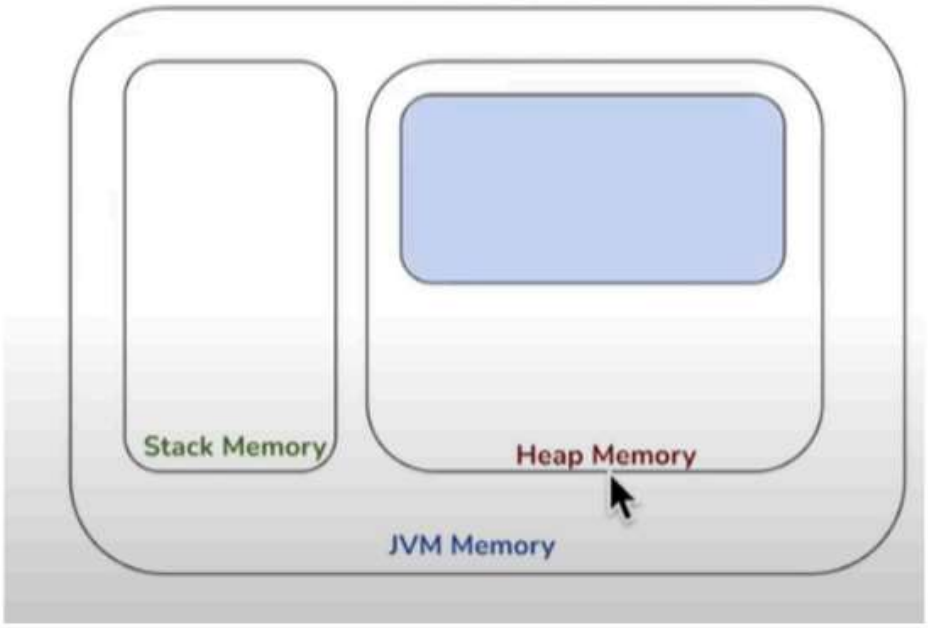
Argument 0: Hello

Argument 1: World

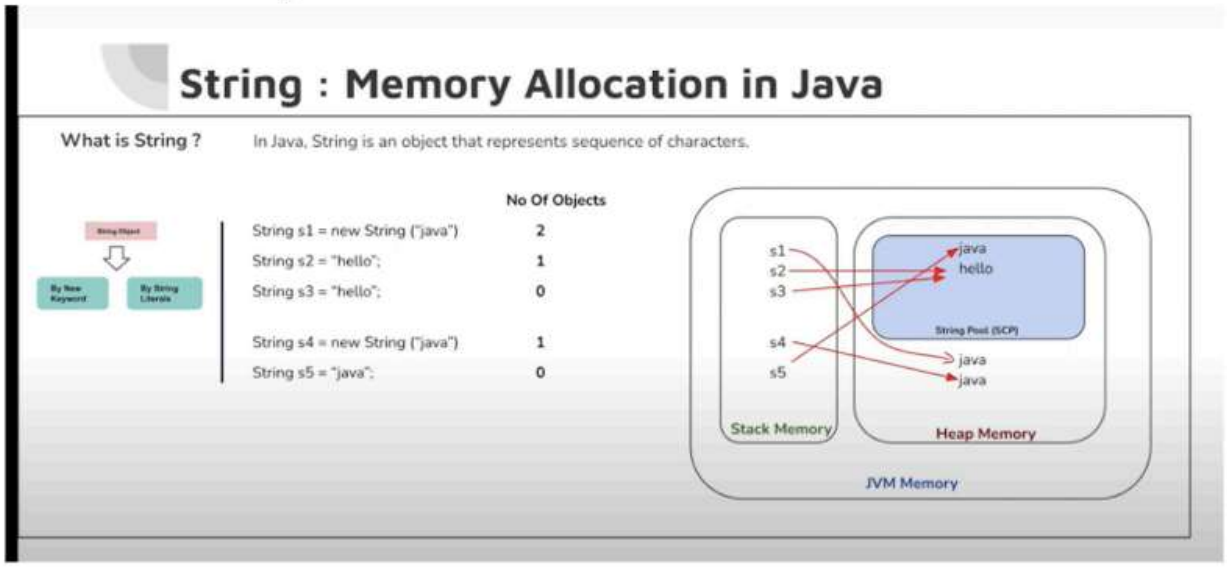
4) Strings in Java

In Java, a **String** is an object that represents a sequence of characters.

JVM has two kinds of storages like heap and stack.



Basically, Heap is used to store the execution of a program.
Stack is used for storage.



String s1 = new String(“java”);

In such a case, JVM will create a new string object in normal (non-pool) **heap memory** and the literal “java” will be placed in the **string constant pool** also. The variable **s1** will

refer to the object in the heap **java** (non-pool).

String s2 = “Hello”;

The value "Hello" is stored in a special area of memory called the **String Pool**, which is part of the **Heap** memory. The variable **s2** will refer to the object in the heap **Hello** in **string constant pool** .

String s3 = “Hello”;

This time JVM will check whether the same object is present in the **string constant pool or not ,** if the object is already there then it will use the same object and doesn’t create an object. The variable **s3** will refer to the object in the heap **Hello** in **string constant pool** if it is present , otherwise it will create a new object and refer to object.

String s4 = new String(“java”);

In such a case, JVM will create a new string object in normal (non-pool) **heap memory** and the literal and JVM will check whether the same object is present in the **string constant pool or not ,** if the object is already there then it will use the same object and doesn’t create an object.
. The variable **s4** will refer to the object in the heap **java** (newly created object) (non-pool).

String s5 = “java”;

This time JVM will check whether the same object is present in the **string constant pool or not ,** if the object is already there then it will use the same object and doesn't create an object. The variable **s5** will refer to the object in the heap **java** in **string constant pool** if it is present , otherwise it will create a new object and refer to the object.

5) Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways.

Polymorphism is an object oriented programming concept that refers to the ability of the method or object to take multiple forms.

Animal dog = new Dog();
Animal cat = new Cat();

2 types

1) Compile time Polymorphism /**Method Overloading** /Static Polymorphism
2) Runtime Polymorphism / **Method Overriding** / Dynamic Polymorphism

1) Method Overloading:

```
package programms;
class Calculator {
	// Overloaded methods
	int add(int a, int b) {
		return a + b;
	}
	double add(double a, double b) {
		return a + b;
	}
}
public class Main {
	public static void main(String[] args) {
		Calculator calc = new Calculator();
		System.out.println(calc.add(3, 5)); // Outputs: 8
		System.out.println(calc.add(2.5, 4.5)); // Outputs: 7.0
```

```
        }
}
```

Q: **Why is method overloading not possible by changing the return type ?**

Method overloading in Java is not possible by changing only the return type because of **ambiguity.**

At the time of calling the method, the return type is not considered by the compiler to determine which method to invoke.

Ex:

```
package programms;
int add(int a, int b) {
        return a + b;
        }
        double add(int a, int b) {
        return a + b;
        }
        public class Main {
                public static void main(String[] args) {
        Main  m= new Main();
        m.add(2, 4); // Which add() should the compiler choose?
        }
        }
```

We can **overload both static and private methods** also.

Ex: **Static**

```
package programms;
class MyClass {
        // Static method with one parameter
        public static void display(int a) {
                System.out.println("Integer: " + a);
        }
        // Static method with a different parameter (overloaded)
        public static void display(String a) {
                System.out.println("String: " + a);
        }
}
```

```
public class Main {
	public static void main(String[] args) {
		// Calling overloaded static methods
		MyClass.display(10); // Calls display(int)
		MyClass.display("Hello"); // Calls display(String)
	}
}
```

Ex2: **Private**

```
package programms;
class MyClass {
	// Private method with one parameter
	private void show(int a) {
		System.out.println("Integer: " + a);
	}
	// Private method with a different parameter (overloaded)
	private void show(String a) {
		System.out.println("String: " + a);
	}
	// Method to test private method overload
	public void testOverloading() {
		show(10); // Calls show(int)
		show("Hello"); // Calls show(String)
	}
}
public class Main {
	public static void main(String[] args) {
		MyClass obj = new MyClass();
		obj.testOverloading(); // Demonstrates private method overloading
	}
}
```

2) Method Overriding:

```
	package programms;
class Animal {
	void sound() {
		System.out.println("Animal makes a sound");
	}
}
class Dog extends Animal {
	@Override
	void sound() {
		System.out.println("Dog barks");
	}
}
class Cat extends Animal {
	@Override
	void sound() {
		System.out.println("Cat meows");
	}
}
public class Main {
	public static void main(String[] args) {
		Animal animal = new Animal();
		animal.sound(); // output : Animal makes a sound
		Animal myAnimal = new Dog(); // Reference of the parent type
		myAnimal.sound(); // Outputs: Dog barks
		Animal myCat = new Cat();
		myCat.sound(); // Outputs: Cat meows
	}
}
```

We **cannot override** static or private methods.

Static methods belong to the class itself, rather than an instance of the class. Since they are associated with the class, they are **not subject to polymorphism**, which is the basis of method overriding in Java.

Private methods are only accessible within the class where they are defined. They cannot be inherited by subclasses, so they cannot be overridden.

6) Constructor

A constructor in Java is a special method that is **used to initialize objects**.
The constructor is **called when an object of a class is created**. It can be **used to set initial values for object attributes**.

A constructor has the **same name as the class** and **does not have a return type** (not even void).

constructors cannot be final, abstract, or static.

Constructors in Java can be **private** or **public**, as well as with other access modifiers like **protected** or **package-private** (default access).

Public Constructor:

- A public constructor allows the class to be instantiated from anywhere in the program. This is the most common type of constructor.

```
public class MyClass {
    public MyClass() {
        // Constructor code
    }
}
```

Private Constructor:

- A private constructor is used to prevent outside classes from creating instances of the class directly. This is often used in design patterns like the Singleton pattern, where you want to control the number of instances of a class.

```
public class MyClass {
    private MyClass() {
        // Constructor code
    }

    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

Protected Constructor:

- A protected constructor allows the class to be instantiated only within the same package or by subclasses.

```
public class MyClass {
   protected MyClass() {
      // Constructor code
   }
}
```

final constructor: Constructors cannot be final because they are not inherited, so there is no need to prevent overriding. The concept of overriding doesn't apply to constructors.

abstract constructor: Constructors cannot be abstract because they are used to initialize objects and cannot be declared without implementation. Abstract methods require subclasses to provide implementation, which doesn't apply to constructors.

static constructor: Constructors cannot be static because they are instance-specific and are called when an object is created, not associated with the class itself. Static members are shared across all instances, but constructors are tied to individual object creation.

Constructors can be **overloaded**, meaning a class can have multiple constructors with different parameter lists.

Even private Constructors also can be overloaded.

An **overload of a private constructor** in Java. Overloading refers to having multiple constructors with the same name but different parameter lists (i.e., different numbers or types of parameters). This is true for private constructors as well, and they can be overloaded just like public constructors.

```
package programms;
class MyClass {
	// Private constructor with no parameters
	private MyClass() {
		System.out.println("Private constructor with no parameters.");
```

```
        }
        // Overloaded private constructor with one parameter
        private MyClass(int a) {
                System.out.println("Private constructor with one parameter: " + a);
        }
        // Overloaded private constructor with two parameters
        private MyClass(String str, int a) {
                System.out.println("Private constructor with two parameters: " + str + " and " +
a);
        }
        // Method to demonstrate constructor overloading
        public static void createObjects() {
                new MyClass();
                new MyClass(10);
                new MyClass("Hello", 20);
        }
}
public class Main {
        public static void main(String[] args) {
                // Call the method to create objects using different private constructors
                MyClass.createObjects();
        }
}
```

Output
Private constructor with no parameters.
Private constructor with one parameter: 10
Private constructor with two parameters: Hello and 20

Constructors cannot override constructors in Java. Constructors are not inherited by subclasses, so the concept of method overriding (which relies on inheritance) does not apply to constructors.

Constructors are not methods: While methods can be inherited and overridden, constructors are special methods used to initialize objects, and they don't participate in the inheritance mechanism in the same way.

Types

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

```
package programms;
//Java Program to create and call a default constructor
class Bike1 {
//creating a default constructor
	Bike1() {
		System.out.println("Bike is created");
	}
//main method
	public static void main(String args[]) {
		Bike1 b = new Bike1(); // calling a default constructor
	}
}
```

Output

```
Bike is created
```

2) **Parameterized Constructor**: Accepts one or more arguments to initialize object fields with specific values.

```
Car car = new Car("Toyota", 2022);
```

Constructors can be **overloaded:**

```
package programms;
class Person {
	String name;
	int age;
	// Default constructor
	public Person() {
		this.name = "Unknown";
		this.age = 0;
		System.out.println("Default constructor called.");
	}
	// Constructor with one parameter
	public Person(String name) {
```

```
        this.name = name;
        this.age = 0;
        System.out.println("Constructor with name called.");
    }
    // Constructor with two parameters
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Constructor with name and age called.");
    }
    // Method to display details
    public void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
    public static void main(String[] args) {
        // Using default constructor
        Person person1 = new Person();
        person1.display();
        // Using constructor with one parameter
        Person person2 = new Person("Alice");
        person2.display();
        // Using constructor with two parameters
        Person person3 = new Person("Bob", 25);
        person3.display();
    }
}
```

Output

```
Default constructor called.
Name: Unknown, Age: 0
Constructor with name called.
Name: Alice, Age: 0
Constructor with name and age called.
Name: Bob, Age: 25
```

Even Private constrcu

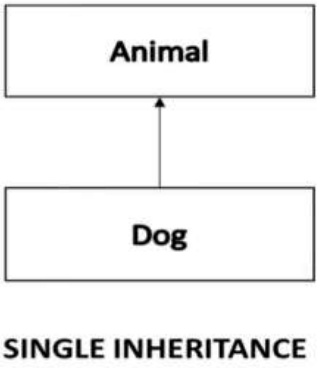
7) Inheritance

What is Inheritance?

Inheritance is a mechanism by which child class acquires the properties and behaviors of a parent class.

Example

```
Class Animal {
    //fields and methods of animal
}

Class Dog extends Animal {
    // Dog gets properties of animal
    //fields and methods
}
```



SINGLE INHERITANCE

The **extends keyword** indicates that you are making a new class that derives from an existing class.

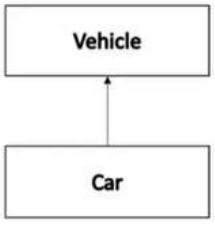
Types

1) **Single**

What is Single Inheritance?

When a class inherits from another class, it is known as a **single inheritance.**

```
public class Vehicle {
    // properties and behaviours
    // of Vehicle class
}

public class Car extends Vehicle {
    // Car gets the properties
    // of Vehicle class
}
```



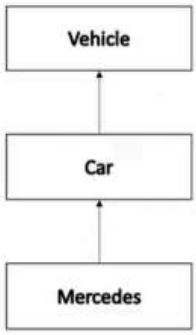
SINGLE INHERITANCE

2) **Multilevel**

What is Multilevel Inheritance?

When there is a chain of inheritance, it is known as **multilevel inheritance.**

```
public class Vehicle {
        // properties and behaviours
        // of Vehicle class
}

public class Car extends Vehicle {
        // Car gets the properties
        // of Vehicle class
}

public class Mercedes extends Car {
        // Mercedes gets the properties
        // of Vehicle and Car classes
}
```



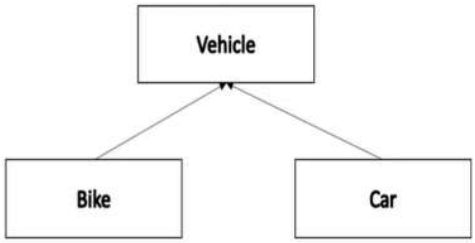
MULTILEVEL INHERITANCE

3) Hierarchical

What is Hierarchical Inheritance?

When two or more classes inherit a single class, it is known as **hierarchical inheritance.**

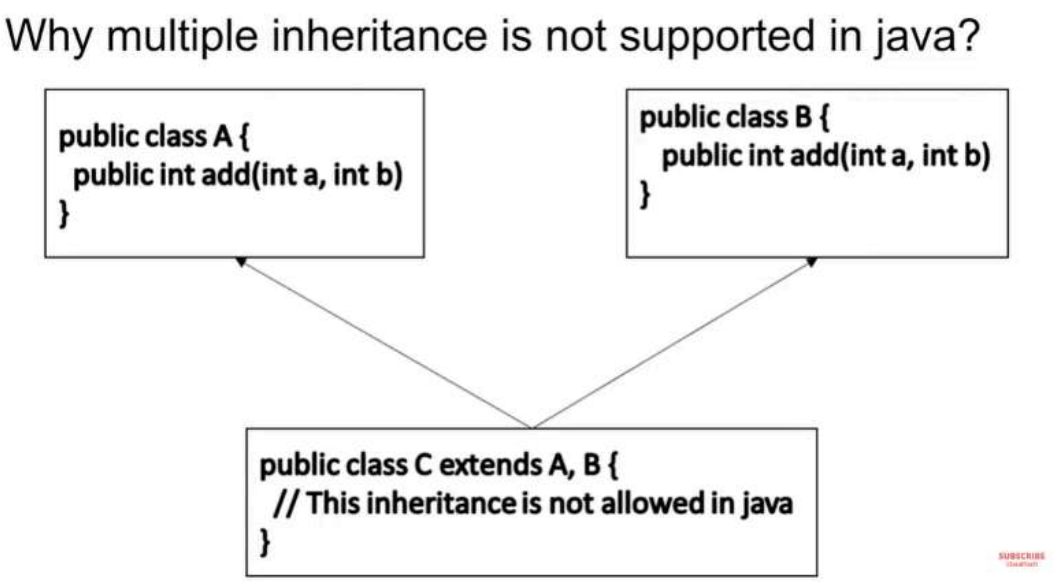
```
public class Vehicle {
        // properties and behaviours
        // of Vehicle class
}

public class Bike extends Vehicle {
        // Bike gets the properties
        // of Vehicle class
}

public class Car extends Vehicle {
        // Car gets the properties
        // of Vehicle
}
```



HIERARCHICAL INHERITANCE

Java does not support **multiple inheritance** (i.e., a class inheriting from more than one class) due to several reasons, the most prominent being **ambiguity** and **maintainability issues.**

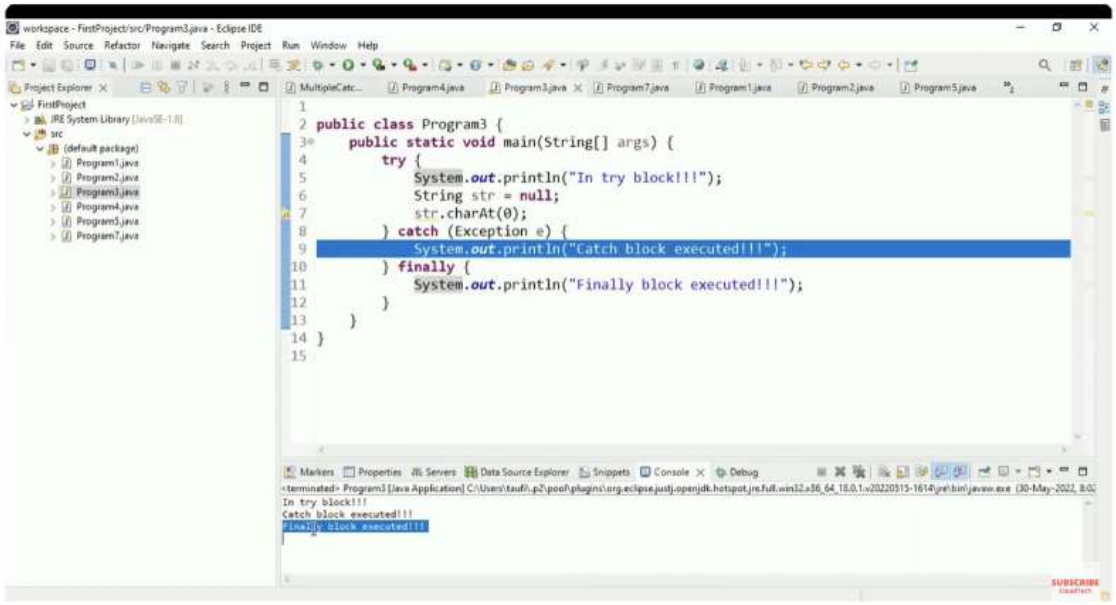
Both Class A, Class B have the same method signature so it leads to ambiguity.



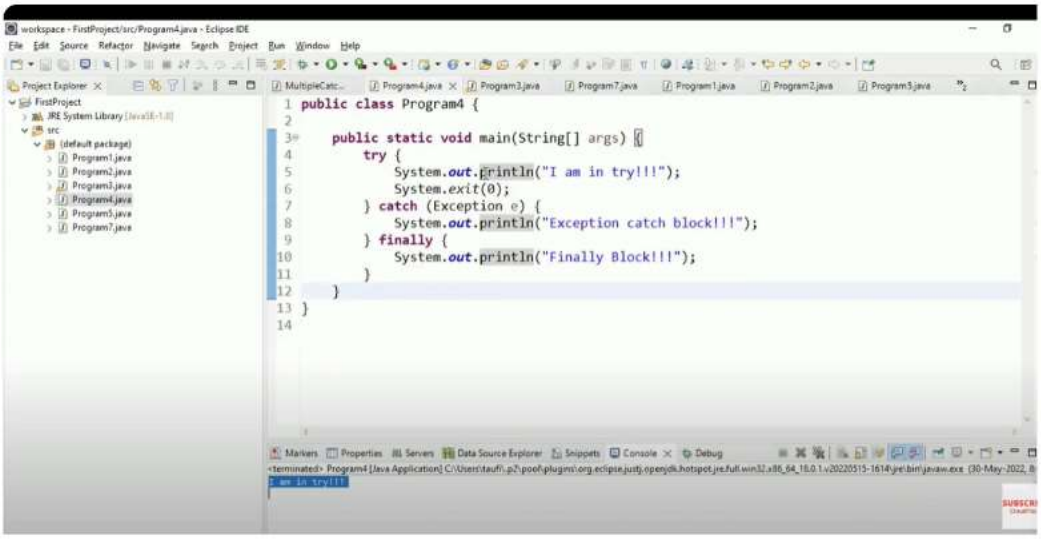
8) Exception Handling

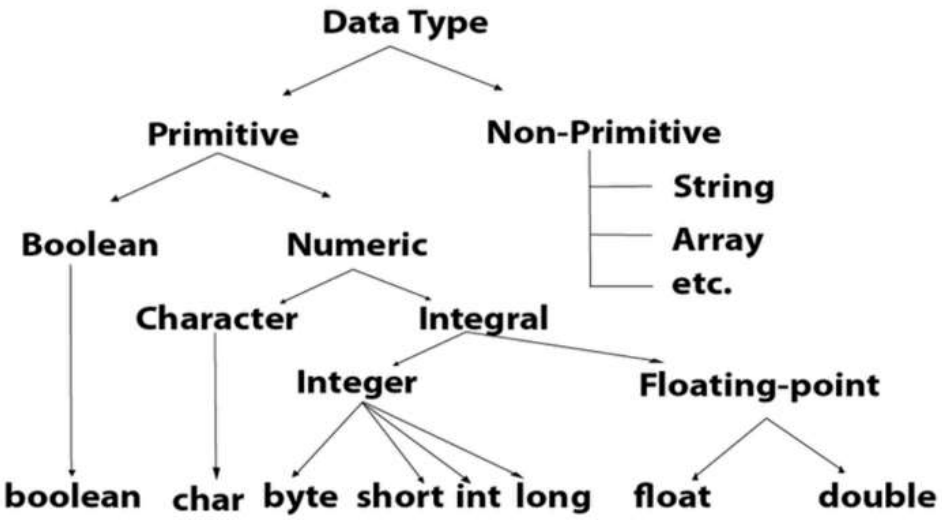
Try
Catch
Finally

Finally class executes whether the exception occurs or not.



But we can also Skip final block using **System.exit(0)**
In the below example, the final block has been skipped.





Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Divide the string into equal parts depending on constant value.

```
package programms;
public class DivideStringTwoEqualParts {
```

```
	public static void main(String[] args) {
		// Input String and parts
		String str = "abcd";
		int parts = 2;
		int length = str.length();
		// Check if the string can be divided equally
		if (length % parts != 0) {
			System.out.println("The string cannot be divided into " + parts + " equal
parts.");
			return;
		}
		// Calculate the size of each part
		int partSize = length / parts;
		// Divide the string and print each part
		for (int i = 0; i < length; i += partSize) {
			System.out.println(str.substring(i, i + partSize));
		}
	}
}
```

Output

```
ab
cd
```

Exs

```
String str = "abcdef";
int parts = 3;
```

Output

```
ab
cd
ef
```

Scanner Class

```
package com.basic;

import java.util.Scanner;

public class ScannerInputExample {
   public static void main(String[] args) {
```

```
// Create a Scanner object for input
Scanner scanner = new Scanner(System.in);

// 1. Reading different data types
System.out.print("Enter an integer: ");
int intValue = scanner.nextInt();
System.out.println("Input Integer: " + intValue);

System.out.print("Enter a floating-point number: ");
float floatValue = scanner.nextFloat();
System.out.println("Input Float: " + floatValue);

System.out.print("Enter a double value: ");
double doubleValue = scanner.nextDouble();
System.out.println("Input Double: " + doubleValue);

System.out.print("Enter a boolean value (true/false): ");
boolean boolValue = scanner.nextBoolean();
System.out.println("Input Boolean: " + boolValue);

// Consume the leftover newline character
            /*
             * After reading a primitive type (e.g., int or float), the Scanner leaves the
             * newline character (\n) in the input buffer. When nextLine() is called
             * afterward, it reads this leftover newline instead of waiting for actual user
             * input.
             *
             * The scanner.next() method reads input until it encounters a delimiter
             * (usually whitespace, such as a space or newline). It does not consume the
             * newline character (\n) at the end of the input. As a result, when you call
             * scanner.nextLine() afterward, it immediately reads the leftover newline
             * instead of waiting for user input.
             */
scanner.nextLine();

System.out.print("Enter a single word (string): ");
String singleWord = scanner.next();
System.out.println("Input Single Word: " + singleWord);

// Consume the leftover newline character again
```

```
        scanner.nextLine();

        System.out.print("Enter a sentence: ");
        String line = scanner.nextLine();
        System.out.println("Input Sentence: " + line);

        // Close the scanner
        scanner.close();
    }
}
```

Extract and print vowels from a given string

```
package programms;
public class VowelsFromString {
        // To initialize a constant char value in Java, you can use the final keyword to
        // declare a constant variable
        public static void main(String[] args) {
                String str = "IceCreAm";
                final char A = 'A';
                final char E = 'E';
                final char I = 'I';
                final char O = 'O';
                final char U = 'U';
                final char a = 'a';
                final char e = 'e';
                final char i = 'i';
                final char o = 'o';
                final char u = 'u';
                int len = str.length();
                for (int j = 0; j < len; j++) {
                        char ch = str.charAt(j);
                        if (ch == A || ch == E || ch == I || ch == O || ch == U || ch == a || ch == e ||
ch == i || ch == o
                                        || ch == u) {
```

```
                        System.out.println(ch);
                        // or
                        // String st = Character.toString(ch); convert char to string.
                        // System.out.println(st);
                    }
                }
                System.out.println(" ");
                // method 2
                String st = "IceCre";
                String vowels = "AEIOUaeiou";
                int length = st.length();
                for (int k = 0; k < length; k++) {
                    char ch1 = st.charAt(k);
                    if (vowels.contains(Character.toString(ch1))) {
                        System.out.println(ch1);
                    }
                }
                System.out.println(" ");
                // method 3
                String stri = "Ice";
                char[] vowelss = { 'A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u' }; // Vowel array
                for (int l = 0; l < stri.length(); l++) {
                    char ch2 = stri.charAt(l);
                    // Check if the current character is in the vowels array
                    for (char vowel : vowelss) {
                        if (ch2 == vowel) {
                            System.out.println(ch2); // Print the vowel
                            // Exit inner loop once a match is found
                        }
                    }
                }
        }
}
```

Output

```
I
e
e
```

A

I
e
e

I
e

Reading or input from user using scanner class

```
package programms;
import java.util.Scanner;
public class ScannerInputExample {
  public static void main(String[] args) {
    // Create a Scanner object for input
    Scanner scanner = new Scanner(System.in);
    // 1. Reading different data types
    System.out.print("Enter an integer: ");
    int intValue = scanner.nextInt();
    System.out.println("Input Integer: " + intValue);
    System.out.print("Enter a floating-point number: ");
    float floatValue = scanner.nextFloat();
    System.out.println("Input Float: " + floatValue);
    System.out.print("Enter a double value: ");
    double doubleValue = scanner.nextDouble();
    System.out.println("Input Double: " + doubleValue);
    System.out.print("Enter a boolean value (true/false): ");
    boolean boolValue = scanner.nextBoolean();
    System.out.println("Input Boolean: " + boolValue);
    // Consume the leftover newline character
                /*
                 * After reading a primitive type (e.g., int or float), the Scanner leaves the
                 * newline character (\n) in the input buffer. When nextLine() is called
                 * afterward, it reads this leftover newline instead of waiting for actual user
                 * input.
                 *
                 * The scanner.next() method reads input until it encounters a delimiter
                 * (usually whitespace, such as a space or newline). It does not consume the
```

```
                * newline character (\n) at the end of the input. As a result, when you call
                * scanner.nextLine() afterward, it immediately reads the leftover newline
                * instead of waiting for user input.
                */
        scanner.nextLine();
        System.out.print("Enter a single word (string): ");
        String singleWord = scanner.next();
        System.out.println("Input Single Word: " + singleWord);
        // Consume the leftover newline character again
        scanner.nextLine();
        System.out.print("Enter a sentence: ");
        String line = scanner.nextLine();
        System.out.println("Input Sentence: " + line);
        // Close the scanner
        scanner.close();
    }
}
```

Palindrome

```
package programms;
import java.util.Scanner;
public class Palindrome {
        public static void main(String[] args) {
                Scanner sc = new Scanner(System.in);
                String input = sc.nextLine(); // Take input from the user
                String pal = method(input);
                // Compare strings using equals method
                if (input.equals(pal)) {
                        System.out.println("palindrome");
                } else {
                        System.out.println("not palindrome");
                }
        }
        // Method to reverse the string
        public static String method(String input) {
                StringBuilder reversed = new StringBuilder();
                // Reverse the string by iterating from the end to the beginning
                for (int i = input.length() - 1; i >= 0; i--) {
```

```
                reversed.append(input.charAt(i));
            }
            return reversed.toString(); // Return the reversed string
        }
}
```

Output
akka
palindrome

Method 2:

```
package programms;
import java.util.Scanner;
public class Palindrome2 {
        public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            String input = sc.nextLine(); // Take input from the user
            StringBuilder reversed = new StringBuilder(input).reverse();
            // Compare the original string with the reversed one
            if (input.equals(reversed.toString())) {
                System.out.println("palindrome");
            } else {
                System.out.println("not palindrome");
            }
        }
}
```

Output
madam
palindrome

Optional class in java8

Every Java Programmer is familiar with NullPointerException. It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in java.util package. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or

alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

Advantages of Optional:

1. **Avoids NullPointerException**: By using Optional, the potential for NullPointerException is minimized.
2. **Better Code Readability**: Makes it clear in the code that a value might be missing (i.e., Optional may or may not contain a value).
3. **Encourages Explicit Null Handling**: Forces the programmer to think about the absence of a value and handle it in a clean way.

public static <T> Optional<T> **empty**()	It returns an empty Optional object. No value is present for this Optional.
public static <T> Optional<T> **of**(T value)	It returns an Optional with the specified present non-null value.
public static <T> Optional<T> **ofNullable**(T value)	It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
public T **get**()	If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
public boolean **isPresent**()	It returns true if there is a value present, otherwise false.

Ex

```
	package com.optional;
import java.util.Optional;
public class OptionalExample {
	public static void main(String[] args) {
		String name = null;
		// It returns an Optional describing the specified value, if non-null, otherwise
		// returns an empty Optional.
		Optional<String> optionalName = Optional.ofNullable(name);
		// Using isPresent() to check if value is not null
		if (optionalName.isPresent()) {
			System.out.println("Name: " + optionalName.get());
		} else {
			System.out.println("Name is null");
		}
	}
}
```

Output

Name is null

Ex2

```
	package com.optional;
import java.util.Optional;
public class OptionalMethodsExample {
	public static void main(String[] args) {
		String name = "John Doe";
		String email = null;
		// Create Optional instances
		Optional<String> optionalName = Optional.ofNullable(name);
		Optional<String> optionalEmail = Optional.ofNullable(email);
		// 1. Using isPresent() to check if the value is present
		if (optionalName.isPresent()) {
			System.out.println("Name is present: " + optionalName.get());
		} else {
			System.out.println("Name is not present");
```

```
                }
                System.out.println(" ");
                // Using isPresent() to check if the value is present
                if (optionalEmail.isPresent()) {
                        System.out.println("email is present: " + optionalEmail.get());
                } else {
                        System.out.println("email is not present");
                }
                System.out.println(" ");
                // 3. Using orElse() to provide a default value if the value is absent
                String defaultName = optionalEmail.orElse("Default Name");
                System.out.println("Name or default: " + defaultName);
                System.out.println(" ");
                // 4. Using map() to transform the value inside Optional
                Optional<String> uppercaseName = optionalName.map(String::toUpperCase);
                uppercaseName.ifPresent(n -> System.out.println("Uppercase name: " + n));
                System.out.println(" ");
                // 5. Using filter() to conditionally check a value inside Optional
                Optional<String> longName = optionalName.filter(n -> n.length() > 5);
                longName.ifPresent(n -> System.out.println("Name is long enough: " + n));
                System.out.println(" ");
                // Demonstrating the result when the value doesn't satisfy the condition
                Optional<String> shortName = optionalName.filter(n -> n.length() < 5);
                shortName.ifPresent(n -> System.out.println("This won't print, name is long"));
                System.out.println(" ");
                // Combining all methods in one flow with chaining
                String result = optionalName.filter(n ->
n.contains("Doe")).map(String::toUpperCase)
                                .orElse("No valid name found");
                System.out.println("Final result: " + result);
        }
}
```

Output

```
Name is present: John Doe
email is not present
Name or default: Default Name
Uppercase name: JOHN DOE
Name is long enough: John Doe
```

Final result: JOHN DOE