JAVA

-James Gosling

Java

Java is an object-oriented and class-based programming language. It’s used as a server-side language for the back-end development of websites, Android apps, and computer software.

Features of Java:

Platform Independent, Secure, Robust (Strong)

Portable, Object oriented Programming Structure

Class

A Class is the blueprint from which individual objects are created.

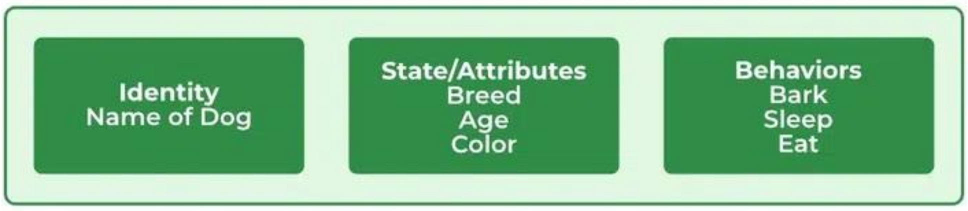
A Class is a collection of fields (data) and methods(fn) that operate on data.

Objects

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class.

An Object consists of:

1. State
2. Behaviour
3. Identity



JVM

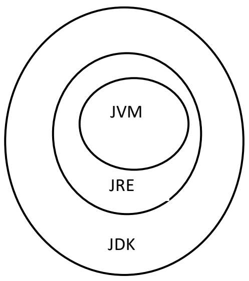
The Java virtual machine (JVM) is a software implementation of a computer that executes programs like a real machine. The Java runtime environment (JRE) consists of the JVM and the Java class libraries and contains the necessary functionality to start Java programs.

JRE

Java Runtime Environment contains JVM, class libraries and other supporting components. As you know the Java source code is compiled into bytecode by Java compiler. This bytecode will be stored in class files. During runtime, this bytecode will be loaded, verified and JVM interprets the bytecode into machine code which will be executed in the machine in which the Java program runs.

A Java Runtime Environment performs the following main tasks respectively.

1. Loads the class – done by Class Loader
2. Verifies the bytecode - done by bytecode verifier
3. Interprets the bytecode - done by JVM



JIT – Just IN Time

This is a component which helps the program execution to happen faster.

Bcoz ,

JIT complier compiles it into native machine code which can be directly executed by the machine in which the Java program runs. Once the byte code is recompiled by JIT compiler, the execution time needed will be much lesser.

Encapsulation

encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class.

<u>Example:</u>

```
public class Sample {
private String name;
 private int age;
 public int getAge() {
 return age;
 }
 public void setAge(int age) {
 this.age = age;
 }
 public String getName() {
 return name;
 }
 public void setName(String name) {
 this.name = name;
 }
}
public class Main {
    public static void main(String[] args) {
        Sample sample = new Sample();
        sample.setName("John");
        sample.setAge(30);
        System.out.println("Name: " + sample.getName());
        System.out.println("Age: " + sample.getAge());
    }
}
```

<u>OUTPUT</u>

```
Name: John
Age: 30
```

Abstraction

Abstraction is a process of hiding the implementation details and showing only functionality to the user. An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

<u>Example</u>

```
abstract class Car
{
 abstract void tagLine();
}
class Honda extends Car
{
 void tagLine()
 {
 System.out.println("Start Something Special");
 }
}
```

```
class Toyota extends Car
{
 void tagLine()
 {
 System.out.println("Drive Your Dreams");
 }
}
public class Main {
    public static void main(String[] args) {
        Car hondaCar = new Honda();
        Car toyotaCar = new Toyota();
        hondaCar.tagLine();   // Output: Start Something Special
        toyotaCar.tagLine();  // Output: Drive Your Dreams
    }
}
```

Inheritance

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.

TYPES

1. Single Inheritance.
2. Multiple Inheritance.
3. Multilevel Inheritance.
4. Hierarchical Inheritance.
5. Hybrid Inheritance.

1. **SINGLE INHERITANCE -** a class can inherit from only one superclass (parent class)

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}

class Dog extends Animal {
   void sound() {
      System.out.println("Dog barks");
   }
}

public class SingleInheritanceExample {
   public static void main(String[] args) {
      Animal animal = new Dog();
      animal.sound();
   }
}
```

   Output

```
Dog barks
```

2. **MULTI INHERITANCE -** a class can inherit from a superclass, and another class can inherit from the derived class, creating a chain of inheritance

```
class Grandparent {
   void display() {
      System.out.println("Grandparent");
   }
}

class Parent extends Grandparent {
   void display() {
        System.out.println("Parent");
    }
}

class Child extends Parent {
    void display() {
        System.out.println("Child");
    }
}

public class MultilevelInheritanceExample {
    public static void main(String[] args) {
        Child child = new Child();
        child.display();
    }
}
```

Output

```
Child
```

3. **HIERARCHICAL INHERITANCE -** multiple classes inherit from a single superclass. All child classes share a common parent class

```
class Vehicle {
    void drive() {
        System.out.println("Vehicle can be driven");
    }
}

class Car extends Vehicle {
    void specificFeature() {
        System.out.println("Car has four wheels");
    }
}

class Motorcycle extends Vehicle {
    void specificFeature() {
        System.out.println("Motorcycle has two wheels");
    }
}

public class HierarchicalInheritanceExample {
    public static void main(String[] args) {
        Car car = new Car();
        Motorcycle motorcycle = new Motorcycle();

        car.drive();
        car.specificFeature();

        motorcycle.drive();
        motorcycle.specificFeature();
    }
}
```

Output

```
Vehicle can be driven
Car has four wheels
Vehicle can be driven
Motorcycle has two wheels
```

4. **HYBRID INHERITANCE -** combination of single, multilevel, hierarchical, and multiple inheritance (through interfaces) in a complex class hierarchy.

```
// Single Inheritance
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Multilevel Inheritance
class Mammal extends Animal {
    void milk() {
        System.out.println("Mammals produce milk");
    }
}

// Hierarchical Inheritance
class Bird extends Animal {
    void fly() {
        System.out.println("Birds can fly");
    }
}

// Multiple Inheritance (via interfaces)
interface Swimmer {
    void swim();
}

class Dolphin extends Mammal implements Swimmer {
    void sound() {
        System.out.println("Dolphin squeaks");
    }

    @Override
    public void swim() {
        System.out.println("Dolphin swims gracefully");
    }
}

public class HybridInheritanceExample {
    public static void main(String[] args) {
        Dolphin dolphin = new Dolphin();

        dolphin.sound(); // Calls overridden method in Dolphin
        dolphin.milk();  // Calls method from Mammal
        dolphin.fly();   // Calls method from Animal
        dolphin.swim();   // Calls swim() from the Swimmer interface
    }
}
```

Output:

```
Dolphin squeaks
Mammals produce milk
Animal makes a sound
Dolphin swims gracefully
```

5. MULTIPLE INHERITANCE

- When one single class is derived from more than one base class.
- Most of the OO languages like java, C# do not support Multiple Inheritance.

POLYMORPHISM

Polymorphism in Java is a concept by which we can ***perform a single action in different ways.*** The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java:

- compile-time polymorphism
- runtime polymorphism

We can perform polymorphism in java by method overloading and method overriding

Method Overloading - multiple methods in the same class have the same name but differ in the number or types of their parameters.

```
class MathOperations {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }
    // Method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }
    // Method to add two doubles
    double add(double a, double b) {
        return a + b;
    }
    // Method to concatenate two strings
    String add(String str1, String str2) {
        return str1 + str2;
    }
}
public class MethodOverloadingExample {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();
        // Overloaded methods are called based on the arguments
provided
        int sum1 = math.add(5, 10);
        int sum2 = math.add(5, 10, 15);
        double sum3 = math.add(2.5, 3.7);
        String concatenatedString = math.add("Hello, ", "World!");
        System.out.println("Sum of two integers: " + sum1);
        System.out.println("Sum of three integers: " + sum2);
        System.out.println("Sum of two doubles: " + sum3);
        System.out.println("Concatenated String: " + concatenatedString);
    }
}
```

Output

```
Sum of two integers: 15
Sum of three integers: 30
Sum of two doubles: 6.2
Concatenated String: Hello, World!
```

Method Overriding - form of run-time (dynamic) polymorphism where a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass must have the same name, return type, and parameters (or a subtype of the parameters) as the method in the superclass.

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class MethodOverridingExample {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal dogAsAnimal = new Dog(); // Polymorphism: Dog object referred to as an Animal reference

        animal.makeSound(); // Calls the makeSound method of the Animal class
        dogAsAnimal.makeSound(); // Calls the overridden makeSound method in the Dog class
    }
}
```

Output

```
Animal makes a sound
Dog barks
```

ACCESS MODIFIER

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

   ```
   class A{
   private int data=40;
   ```

```
private void msg(){System.out.println("Hello java");}
}
  -------------
public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data); //Compile Time Error
   obj.msg(); //Compile Time Error
   }
}
```

2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
---------------------
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();   //Compile Time Error
   obj.msg();  //Compile Time Error
  }
}
```

3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
----------------------------------
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Output: Hello

4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

```
//save by A.java
  package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
------------------
//save by B.java
  package mypack;
```

```
import pack.*;
  class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output: Hello

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

CONSTRUCTOR

constructor is a special type of method that is used to initialize objects when they are created. Constructors are called automatically when an object of a class is instantiated. Constructors have the same name as the class and do not have a return type.

Types:

1. Default / Non – Parameterized Construction
2. Parameterized Construction
3. Copy Construction

1. **Default / Non – Parameterized Construction**

```
class Student {
    String name;
    int age;
    String course;

    // Default constructor
    Student() {
        // Initialize default values
        name = "John Doe";
        age = 20;
        course = "Testing";
    }

    void displayDetails() {
        System.out.println("Student Name: " + name);
        System.out.println("Student Age: " + age);
        System.out.println("Course: " + course);
    }
}

public class DefaultConstructorExample {
    public static void main(String[] args) {
        // Creating a Student object using the default constructor
        Student student = new Student();

        // Displaying student details
        student.displayDetails();
    }
}
```

Output:

```
Student Name: John Doe
Student Age: 20
Course: Testing
```

2. Parameterized Construction

```
class Student {
    String name;
    int age;
    String course;

    // Parameterized constructor
    Student(String name, int age, String course) {
        this.name = name;
        this.age = age;
        this.course = course;
    }

    void displayDetails() {
        System.out.println("Student Name: " + name);
        System.out.println("Student Age: " + age);
        System.out.println("Course: " + course);
    }
}

public class ParameterizedConstructorExample {
    public static void main(String[] args) {
        // Creating a Student object using the parameterized
constructor
        Student student = new Student("Alice", 22, "Computer
Science");
        // Displaying student details
        student.displayDetails();
    }
}
```

Output:

```
Student Name: Alice
Student Age: 22
Course: Computer Science
```

3. Copy Constructor

```
class Person {
    String name;
    int age;

    // Parameterized constructor
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Copy constructor
    Person(Person other) {
        this.name = other.name;
        this.age = other.age;
    }

    void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class CopyConstructorExample {
    public static void main(String[] args) {
        // Creating a Person object using the parameterized constructor
        Person person1 = new Person("Alice", 25);

        // Creating a new Person object by copying the state of person1
        Person person2 = new Person(person1);

        // Displaying details of both persons
        System.out.println("Person 1:");
        person1.displayDetails();

        System.out.println("\nPerson 2 (Copy of Person 1):");
        person2.displayDetails();
    }
}
```

Output:

```
Person 1:
Name: Alice
Age: 25

Person 2 (Copy of Person 1):
Name: Alice
Age: 25
```

Array Concept - collection of similar type of elements

Types:

1. **Single Dimensional Array**

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

```
Output:
10
20
70
40
50
```

2. Multidimensional Array

```
public class MultiDimensionalArrayExample {
    public static void main(String[] args) {
        // Creating a 2D array (matrix) of integers
        int[][] matrix = new int[3][3];     // 3x3 matrix

        // Initializing the elements of the 2D array
        matrix[0][0] = 1;
        matrix[0][1] = 2;
        matrix[0][2] = 3;
        matrix[1][0] = 4;
        matrix[1][1] = 5;
        matrix[1][2] = 6;
        matrix[2][0] = 7;
        matrix[2][1] = 8;
        matrix[2][2] = 9;

        // Displaying the 2D array (matrix)
        System.out.println("2D Array (Matrix):");
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
2D Array (Matrix):
1 2 3
4 5 6
7 8 9
```

Java Program to print the array elements using **for-each loop**

```
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
}}
```

Output:

```
33
3
4
5
```

String Methods

1. **length():** Returns the length (number of characters) of the string.

2. **charAt(int index):** Returns the character at the specified index.

3. **concat(String str):** Concatenates the specified string to the end of this string.

4. **equals(Object obj):** Compares this string to the specified object to check if they are equal.

5. **equalsIgnoreCase(String anotherString):** Compares this string to another string, ignoring case differences.

6. **compareTo(String anotherString):** Compares two strings lexicographically and returns an integer.

7. **compareToIgnoreCase(String str):** Compares two strings lexicographically, ignoring case differences.

8. **startsWith(String prefix):** Checks if the string starts with the specified prefix.

9. **endsWith(String suffix):** Checks if the string ends with the specified suffix.

10. **indexOf(int ch):** Returns the index within the string of the first occurrence of the specified character.

11. **indexOf(int ch, int fromIndex):** Returns the index within the string of the first occurrence of the specified character, starting from the specified index.

12. **indexOf(String str):** Returns the index within the string of the first occurrence of the specified substring.

13. **indexOf(String str, int fromIndex):** Returns the index within the string of the first occurrence of the specified substring, starting from the specified index.

14. **lastIndexOf(int ch):** Returns the index within the string of the last occurrence of the specified character.

15. **lastIndexOf(int ch, int fromIndex):** Returns the index within the string of the last occurrence of the specified character, searching backward from the specified index.

16. **lastIndexOf(String str):** Returns the index within the string of the last occurrence of the specified substring.

17. **lastIndexOf(String str, int fromIndex):** Returns the index within the string of the last occurrence of the specified substring, searching backward from the specified index.

18. **substring(int beginIndex):** Returns a substring of the string, starting from the specified index.

19. **substring(int beginIndex, int endIndex):** Returns a substring of the string, starting from the specified begin index and ending at the specified end index.

20. **toLowerCase():** Converts all characters in the string to lowercase.

21. **toUpperCase():** Converts all characters in the string to uppercase.

22. **trim():** Removes leading and trailing whitespace from the string.

23. **replace(char oldChar, char newChar):** Replaces all occurrences of oldChar with newChar in the string.

24. **replace(CharSequence target, CharSequence replacement):** Replaces all occurrences of target with replacement in the string.

25. **split(String regex):** Splits the string into an array of substrings based on the provided regular expression.

26. **startsWith(String prefix, int toffset):** Checks if the string starts with the specified prefix, beginning at the specified index.

27. **endsWith(String suffix):** Checks if the string ends with the specified suffix.

28. **isEmpty():** Returns true if the string is empty (has a length of 0).

29. **contains(CharSequence sequence):** Checks if the string contains the specified character sequence.

30. **matches(String regex):** Determines if the string matches the given regular expression.

31. **getBytes():** Encodes the string into a sequence of bytes using the platform's default character set.

Example

```
public class StringMethodsExample {
   public static void main(String[] args) {
      String text = "   Hello, World!   ";

      // charAt(int index)
      char character = text.charAt(7);
      System.out.println("charAt(7): " + character);

      // length()
      int length = text.length();
```

```
        System.out.println("length(): " + length);

        // substring(int beginIndex)
        String substring1 = text.substring(5);
        System.out.println("substring(5): " + substring1);

        // substring(int beginIndex, int endIndex)
        String substring2 = text.substring(7, 12);
        System.out.println("substring(7, 12): " + substring2);

        // toLowerCase()
        String lowercase = text.toLowerCase();
        System.out.println("toLowerCase(): " + lowercase);

        // toUpperCase()
        String uppercase = text.toUpperCase();
        System.out.println("toUpperCase(): " + uppercase);

        // trim()
        String trimmed = text.trim();
        System.out.println("trim(): " + trimmed);

        // indexOf(String str)
        int indexOfWorld = text.indexOf("World");
        System.out.println("indexOf(\"World\"):         "         +
indexOfWorld);

        // contains(CharSequence sequence)
        boolean containsHello = text.contains("Hello");
        System.out.println("contains(\"Hello\"):         "         +
containsHello);

        // startsWith(String prefix)
        boolean startsWithHello = text.startsWith("Hello");
        System.out.println("startsWith(\"Hello\"):  "  +
    startsWithHello);

        // endsWith(String suffix)
        boolean endsWithSpace = text.endsWith(" ");
        System.out.println("endsWith(\"  \"):  "  +
    endsWithSpace);

        //  replace(CharSequence  target,  CharSequence
    replacement)
        String replacedText = text.replace("World", "Java");
        System.out.println("replace(\"World\",  \"Java\"):  "  +
    replacedText);

        // split(String regex)
        String[] words = text.split(" ");
        System.out.println("split(\" \"): ");
        for (String word : words) {
            System.out.println(word);
        }
      }
    }
```

Output:

```
charAt(7): W
length(): 16
substring(5):  Hello, World!
substring(7, 12): World
toLowerCase():    hello, world!
toUpperCase():    HELLO, WORLD!
trim(): Hello, World!
indexOf("World"): 7
contains("Hello"): true
startsWith("Hello"): false
```

```
endsWith(" "): true
replace("World", "Java"):    Hello, Java!
split(" "):
Hello,
,
,
World!
```

LOOPS / Control Statements

1. **Selection Control Statements**
   - If
   - If .. else
   - Nested if..else ..if
   - Switch case
2. **Iteration Control Statements**
   - for
   - while
   - do-while
3. **Jump Control Statements**
   - Return
   - Break
   - Continue

If-else Loop:

```
import java.util.Scanner;

public class EligibilityCheck {

    public static void main(String[] args) {

        // Create a Scanner object to read input from the user

        Scanner scanner = new Scanner(System.in);

        // Prompt the user to enter their age
        System.out.print("Enter your age: ");
        int age = scanner.nextInt();
        // Close the scanner to prevent resource leak
        scanner.close();
        // Check eligibility to vote
        if (age >= 18) {
            System.out.println("You are eligible to vote.");
        } else {
            System.out.println("You are not eligible to vote.");
        }
    }}
```

Output:

```
Enter your age: 25
You are eligible to vote.
```

Nested If:

```
import java.util.Scanner;
public class IfStatementExample {
    public static void main(String[] args) {
        // Create a Scanner object to read input from the user
        Scanner scanner = new Scanner(System.in);

        // Prompt the user to enter a number
        System.out.print("Enter a number: ");
        int number = scanner.nextInt();
        // Close the scanner to prevent resource leak
        scanner.close();

        // Check if the number is positive, negative, or zero
        if (number > 0) {
            System.out.println("The number is positive.");
        } else if (number < 0) {
            System.out.println("The number is negative.");
        } else {
            System.out.println("The number is zero.");
        }}}
```

Output

```
Enter a number: 5
The number is positive.
```

For loop:

```
public class ForLoopExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Number: " + i);
        }
    }
}
```

Output:

```
Number: 1
```

```
Number: 2
Number: 3
Number: 4
Number: 5
```

While Loop:

```
public class WhileLoopExample {
    public static void main(String[] args) {
        int i = 1; // Initialize the loop control variable

        // Use a while loop to count from 1 to 5
        while (i <= 5) {
            System.out.println("Number: " + i);
            i++; // Increment the loop control variable
        }
    }
}
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Do-While loop:

```
import java.util.Scanner;
public class MultiplicationTable {
    public static void main(String[] args) {
        // Create a Scanner object to read input from the user
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int number = sc.nextInt();
        sc.close();

        int i = 1; // Initialize the loop control variable

  System.out.println("Multiplication Table for " + number + ":");
            do {
    System.out.println(number + " x " + i + " = " + (number * i));
            i++; // Increment the loop control variable
        } while (i <= 5); // Print the table from 1 to 5
    }
}
```

Output

```
Enter a number: 5
Multiplication Table for 5:
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
```

Break:

```
public class BreakStatementExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                System.out.println("Reached the break statement.");
                break; // Exit the loop when i is 5
            }
            System.out.println("Current value of i: " + i);
        }
        System.out.println("Loop finished.");
    }
}
```

```
Output:
Current value of i: 1
Current value of i: 2
Current value of i: 3
Current value of i: 4
Reached the break statement.
Loop finished.
```

Continue:

```
public class ContinueStatementExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                System.out.println("Skipping iteration " + i);
                continue; // Skip this iteration and go to the next one
            }
            System.out.println("Current value of i: " + i);
        }
        System.out.println("Loop finished.");
    }}
```

```
Output:
Current value of i: 1
Current value of i: 2
Skipping iteration 3
Current value of i: 4
Current value of i: 5
Loop finished.
```

Exception Handling

Exception handling in Java is a mechanism that allows you to handle unexpected or exceptional situations that may occur during program execution. Exceptions are runtime errors or exceptional conditions that disrupt the normal flow of a program. Java provides a robust exception handling mechanism to deal with these situations, ensuring that your program can gracefully recover from errors or handle them in an appropriate manner.

Key concepts and components of exception handling in Java include:

Exception Classes: Java has a hierarchy of exception classes, with java.lang.Throwable as the root class. Exceptions can be categorized into two main types:

Checked Exceptions: These are exceptions that are checked at compile-time and must be either caught using try-catch blocks or declared in the method signature using the **throws** keyword. Examples include *IOException and SQLException.*
Unchecked Exceptions (Runtime Exceptions): These are exceptions that are not checked at compile-time and can occur at runtime. They do not require explicit handling. Examples include *NullPointerException and ArrayIndexOutOfBoundsException.*
try-catch Blocks: You can use try and catch blocks to handle exceptions. The try block contains the code that may throw an exception, and the catch block contains code to handle the exception if it occurs. Multiple catch blocks can be used to handle different types of exceptions.

```
try {
   // Code that may throw an exception
} catch (ExceptionType1 e1) {
   // Handle ExceptionType1
} catch (ExceptionType2 e2) {
   // Handle ExceptionType2
} finally {
    // Code that always executes, whether an exception occurs or
not
}
```

throws Keyword: declare exceptions in the method signature using the throws keyword. This indicates that the method may throw certain exceptions, and it's the responsibility of the calling code to handle them.

```
public void myMethod() throws SomeException {
    // Method code that may throw SomeException
}
```

throw Keyword: throw keyword to manually throw exceptions. This is often used in custom exception handling scenarios.

```
if (someCondition) {
    throw new CustomException("This is a custom exception.");
}
```

finally Block: The finally block is optional and is used to specify code that should always execute, whether an exception occurs or not. It is typically used for resource cleanup (e.g., closing files or network connections).

```
try {
    // Code that may throw an exception
} catch (Exception e) {
    // Handle the exception
} finally {
    // Code that always executes
}
```

Example Program

```
import java.util.Scanner;

public class ExceptionHandlingExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int result = 0;

        try {
            System.out.print("Enter numerator: ");
            int numerator = scanner.nextInt();

            System.out.print("Enter denominator: ");
            int denominator = scanner.nextInt();

            result = numerator / denominator; // Possible division by
zero exception

            System.out.println("Result of division: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero is not
allowed.");
        } catch (Exception e) {
            System.out.println("An unexpected error occurred: " +
e.getMessage());
        } finally {
            System.out.println("Finally block always executes.");
            scanner.close(); // Close the scanner in the finally block
        }

        System.out.println("Program continues after exception
handling.");
    }
}
```

Output:

```
Enter numerator: 10
Enter denominator: 0
Error: Division by zero is not allowed.
Finally block always executes.
Program continues after exception handling.
```

Example 2:

```
public class ArrayIndexOutOfBoundsExceptionExample {
    public static void main(String[] args) {
        int[] numbers = { 1, 2, 3, 4, 5 };

        try {
            // Attempt to access an element outside the valid index
range
            int invalidValue = numbers[10]; // This will throw an
ArrayIndexOutOfBoundsException
            System.out.println("Value at index 10: " + invalidValue);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught an Array Index Out Of
Bounds Exception: " + e.getMessage());
        }

        System.out.println("Program continues after exception
handling.");
    }
}
```

Output:

```
Caught an ArrayIndexOutOfBoundsException: Index 10 out of
bounds for length 5
Program continues after exception handling.
```

Java Collection

A "collection" refers to a framework or group of classes and interfaces in the Java Collections Framework (located in the java.util package) that are used to store, manipulate, and manage groups of objects, often referred to as elements or items. Java collections are used to work with data structures that can dynamically grow or shrink in size, making it easier to handle and process collections of data.

Collections Framework: The Java Collections Framework provides a standardized way to work with various types of collections, including lists, sets, and maps. It includes interfaces, classes, and algorithms that help developers efficiently manage and manipulate collections of objects.

Interfaces:

These interfaces define common methods and behaviors that various collection types should adhere to. Some of the primary collection interfaces include:

- **List:** An ordered collection that allows duplicate elements (e.g., ArrayList, LinkedList).
- **Set:** A collection that does not allow duplicate elements (e.g., HashSet, TreeSet).
- **Map:** A collection of key-value pairs (e.g., HashMap, TreeMap).

Classes: Concrete implementations of the collection interfaces are provided as classes. These classes offer specific behaviors and optimizations for different use cases. For example:

- **ArrayList**: A dynamic array that can grow in size.
- **HashSet:** A set that uses a hash table for efficient element lookup.

- **HashMap:** A map that uses a hash table for key-value pairs.

Algorithms: The framework includes various algorithms for performing common operations on collections, such as sorting, searching, and filtering. These algorithms are often available as static methods in the *Collections* and *Arrays* classes.

Generics: Generics are extensively used in Java collections to provide type safety. They allow you to specify the type of elements that a collection can hold.

Iterators: Collections in Java often provide iterators, which are used to traverse through the elements of a collection sequentially.

Example – ArrayList:

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Create an ArrayList of strings
        ArrayList<String> fruits = new ArrayList<>();

        // Add elements to the ArrayList
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Date");

        // Display the ArrayList elements
        System.out.println("ArrayList elements:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }
        // Get the size of the ArrayList
        int size = fruits.size();
        System.out.println("\nSize of the ArrayList: " + size);

        // Check if the ArrayList contains a specific element
        boolean containsBanana = fruits.contains("Banana");
        System.out.println("\nArrayList contains 'Banana': " +
containsBanana);

        // Remove an element from the ArrayList
        fruits.remove("Cherry");

        // Display the modified ArrayList
        System.out.println("\nModified ArrayList:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

Output:

```
ArrayList elements:
Apple
Banana
Cherry
Date

Size of the ArrayList: 4

ArrayList contains 'Banana': true

Modified ArrayList:
Apple
Banana
```

Date

Example – Hash Set

```
import java.util.HashSet;
public class HashSetExample {
    public static void main(String[] args) {
        // Create a HashSet of integers
        HashSet<Integer> numbers = new HashSet<>();

        // Add elements to the HashSet
        numbers.add(5);
        numbers.add(10);
        numbers.add(15);
        numbers.add(20);
        numbers.add(25);

        // Display the HashSet elements
        System.out.println("HashSet elements:");
        for (int number : numbers) {
            System.out.println(number);
        }

        // Check if the HashSet contains a specific element
        boolean contains15 = numbers.contains(15);
        System.out.println("\nHashSet contains 15: " +
contains15);

        // Remove an element from the HashSet
        numbers.remove(10);

        // Display the modified HashSet
        System.out.println("\nModified HashSet:");
        for (int number : numbers) {
            System.out.println(number);
        }

        // Get the size of the HashSet
        int size = numbers.size();
        System.out.println("\nSize of the HashSet: " + size);
    }
}
```

Output:

```
HashSet elements:
5
10
15
20
25

HashSet contains 15: true

Modified HashSet:
5
15
20
25

Size of the HashSet: 4
```

Example - HashMap

```
import java.util.HashMap;
public class HashMapExample {
    public static void main(String[] args) {
        // Create a HashMap to store key-value pairs of countries
and their capitals
        HashMap<String, String> capitals = new HashMap<>();

        // Add key-value pairs to the HashMap
        capitals.put("USA", "Washington, D.C.");
        capitals.put("Canada", "Ottawa");
        capitals.put("France", "Paris");
        capitals.put("Germany", "Berlin");
        capitals.put("Japan", "Tokyo");

        // Retrieve and display the capital of a specific country
        String capitalOfFrance = capitals.get("France");
        System.out.println("Capital of France: " +
capitalOfFrance);

        // Check if the HashMap contains a specific key
        boolean containsCanada = capitals.containsKey("Canada");
        System.out.println("Contains Canada: " + containsCanada);

        // Check if the HashMap contains a specific value
        boolean containsBeijing =
capitals.containsValue("Beijing");
        System.out.println("Contains Beijing: " + containsBeijing);

        // Display all countries and their capitals
        System.out.println("\nAll countries and capitals:");
        for (String country : capitals.keySet()) {
            String capital = capitals.get(country);
            System.out.println(country + ": " + capital);
        }
        // Remove a key-value pair from the HashMap
        capitals.remove("Germany");

        // Display the modified HashMap
        System.out.println("\nModified countries and capitals:");
        for (String country : capitals.keySet()) {
            String capital = capitals.get(country);
            System.out.println(country + ": " + capital);
        }
        // Get the size of the HashMap
        int size = capitals.size();
        System.out.println("\nSize of the HashMap: " + size);
    }
}
```

Output:

```
Capital of France: Paris
Contains Canada: true
Contains Beijing: false

All countries and capitals:
Germany: Berlin
Canada: Ottawa
France: Paris
Japan: Tokyo
USA: Washington, D.C.

Modified countries and capitals:
Canada: Ottawa
France: Paris
Japan: Tokyo
USA: Washington, D.C.

Size of the HashMap: 4
```