Java Week - 2

1. Generics

Generics provide type safety and code reusability by allowing a class, interface, or method to operate on different data types while ensuring type correctness at compile-time.

Benefits of Generics:

- **Type Safety**: Prevents runtime ClassCastException by enforcing type checks at compile-time.
- **Code Reusability**: Allows a single class or method to work with different data types.
- **Eliminates Type Casting**: No need for explicit casting.

Generic Classes

A generic class is a class that has one or more type parameters. Here's a simple example of a generic class. In this example, T is a type parameter. You can create instances of Box for different types.

```
class Box<T> {

   private T value;

   public void setValue(T value) {

      this.value = value;

   }

   public T getValue() {

      return value;

   }

}



public class Main {

   public static void main(String[] args) {

      Box<Integer> intBox = new Box<>();

      intBox.setValue(10);
```

```
        System.out.println("Integer Value: " + intBox.getValue());

        Box<String> strBox = new Box<>();

        strBox.setValue("Hello");

        System.out.println("String Value: " + strBox.getValue());

    }

}
```

Generic Methods

You can also create generic methods within non-generic classes. You can use this method with different types. Here's an example,

```
public class Util {

    public <E> void printArray(E[] array) {

        for (E element : array) {

            System.out.print(element + " ");

        }

        System.out.println();

    }

    public static void main(String[] args) {

        Integer[] intArray = {1, 2, 3, 4, 5};

        String[] stringArray = {"apple", "banana", "orange"};

        Util util = new Util();

        util.printArray(intArray);

        util.printArray(stringArray);

    }

}
```

Wildcard in Generics

Wildcards (?) allow us to create more flexible generic methods and classes when the exact type is unknown.

```
public class Printer {
  public static void printList(List<?> list) {
    for (Object item : list) {
      System.out.print(item + " ");
    }
    System.out.println();
  }
  public static void main(String[] args) {
    List<Integer> intList = Arrays.asList(1, 2, 3);
    List<String> stringList = Arrays.asList("apple", "banana", "orange");
    Printer.printList(intList);
    Printer.printList(stringList);
  }
}
```

2. Functional Interfaces & Lambda Expressions

A functional interface is an interface that contains one and only one abstract method. It is a way to define a contract for behavior as an argument to a method invocation.

A lambda expression is a block of code that gets passed around, like an anonymous method. It is a way to pass behavior as an argument to a method invocation and to define a method without a name.

The basic syntax of a lambda expression consists of the parameter list, the arrow (->), and the body. The body can be either an expression or a block of statements.

(parameters) -> expression

(parameters) -> { statements }

Parameter List: This represents the parameters passed to the lambda expression. It can be empty or contain one or more parameters enclosed in parentheses. If there's only one parameter and its type is inferred, you can omit the parentheses.

Arrow Operator (->): This separates the parameter list from the body of the lambda expression.

Lambda Body: This contains the code that makes up the implementation of the abstract method of the functional interface. The body can be a single expression or a block of code enclosed in curly braces.

Example:

```
interface MathOperation {

    int operate(int a, int b);

}

public class LambdaExample {

    public static void main(String[] args) {

        MathOperation addition = (a, b) -> a + b;

        System.out.println("Sum: " + addition.operate(10, 5));

    }

}
```

3. Streams API

A stream in Java is simply a wrapper around a data source, allowing us to perform bulk operations on the data in a convenient way. The Java Stream API, introduced in Java 8, is a powerful abstraction for processing sequences of elements, such as collections or arrays, in a functional and declarative way. Streams are designed to be used in a chain of operations, allowing you to create complex data processing pipelines.

Creating Streams:

Creating a Stream from an Array, example:

```
List<String> fruits = Arrays.asList("Apple", "Banana", "Orange", "Mango");

// Creating a stream from a collection
```

```
Stream<String> fruitStream = fruits.stream();
```

Intermediate Operations:

These operations transform a stream into another stream and do not produce a final result. They are lazy, meaning they are executed only when a terminal operation is applied.

a) filter()

Used to select elements based on a condition. Returns a new stream with elements that satisfy the condition.

Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

names.stream()

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

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

b) map()

Transforms each element in the stream. Used for converting one type to another.

Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> squaredNumbers = numbers.stream()

    .map(n -> n * n)

    .collect(Collectors.toList());

System.out.println(squaredNumbers);
```

c) sorted()

Sorts elements in natural order (or custom comparator).

Example:

```
List<String> names = Arrays.asList("John", "Alice", "Bob");

names.stream()
```

```
    .sorted()
    .forEach(System.out::println);
```

Terminal Operations

Terminal operations trigger the processing of elements and produce a result or a side effect. They are the final step in a stream pipeline. They are eager, which means that they are executed immediately.

a) forEach()

Iterates over each element and applies an action.

Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream()
    .forEach(System.out::println);
```

b) collect()

Gathers the result into a List, Set, or Map. Commonly used with Collectors.toList(), Collectors.toSet(), and Collectors.toMap().

Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> upperCaseNames = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
System.out.println(upperCaseNames);
```

c) reduce()

Performs aggregation (sum, min, max, etc.). Takes an identity value and a binary operator.

Example: (Summing all elements in a list)

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()

    .reduce(0, Integer::sum);

System.out.println("Sum: " + sum);
```

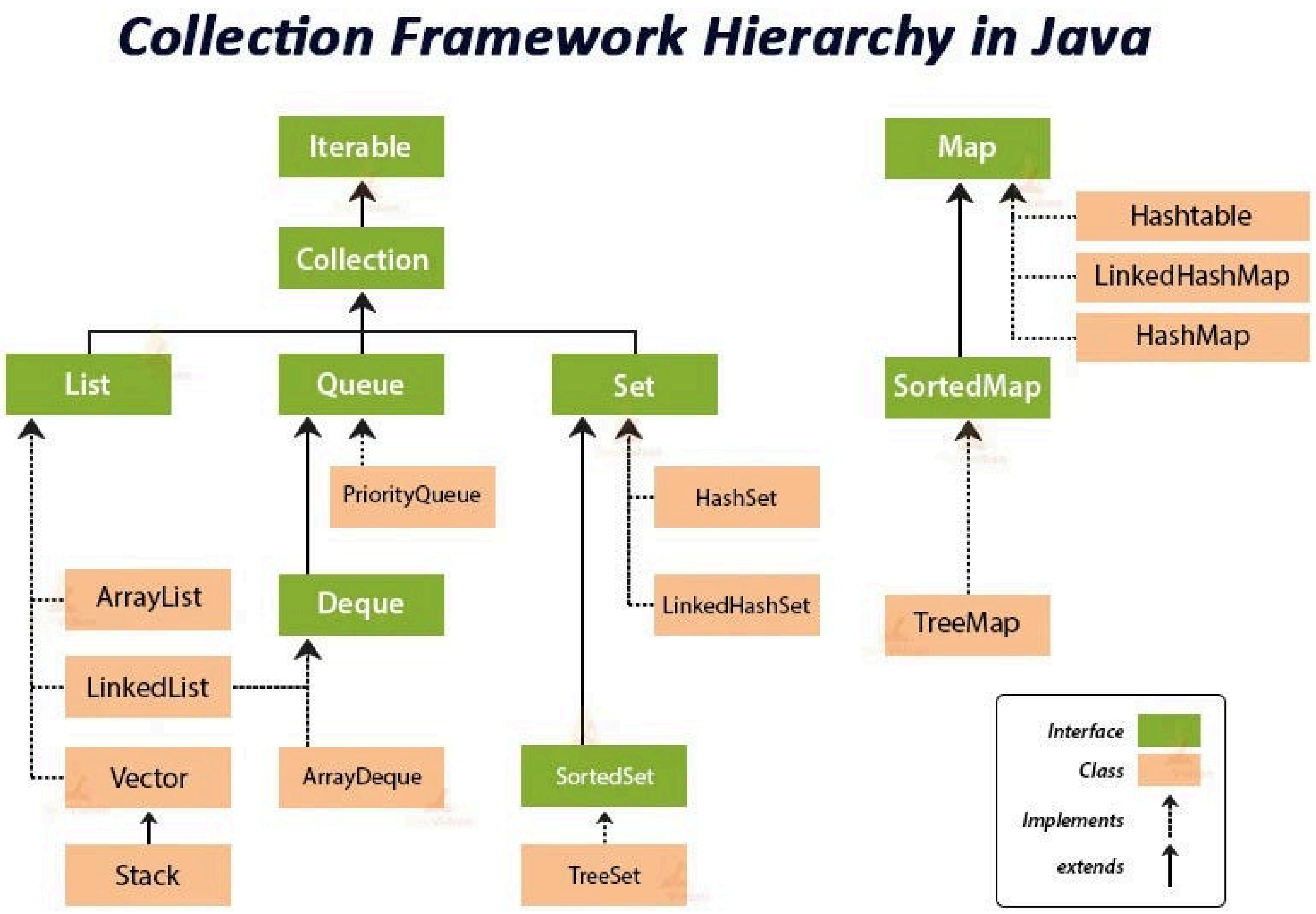
4. Collections Framework

Java Collections Framework provides a set of interfaces and classes to store and manipulate groups of objects. Collections make it easier to work with groups of objects, such as lists, sets, and maps. Collection is the root interface of the Java Collections Framework, representing a group of objects. Its key sub-interfaces include **List, Set, and Queue**.

Benefits of Using Collections

- **Dynamic Sizing:** Collections grow and shrink dynamically compared to arrays.
- **Reusable Algorithms:** Provides built-in sorting, searching, and iteration methods.
- **Efficient Performance:** Optimized data structures for different use cases.

Hierarchy of Java Collections



List

An ordered collection that allows duplicate elements. Common implementations include **ArrayList, LinkedList, and Vector**.

Example:

```
public class ListExample {

  public static void main(String[] args) {

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

    list.add("Apple");

    list.add("Banana");

    list.add("Apple"); // Allows duplicates

    System.out.println(list); // Output: [Apple, Banana, Apple]

  }
}
```

Queue

An interface that extends **Collection**, primarily used for handling elements in a **FIFO (first-in-first-out)** manner. However, exceptions exist, such as **PriorityQueue**, which orders elements based on a comparator or their natural ordering. Implementations include **ArrayDeque, LinkedList, and PriorityQueue**.

Example:

```
public class QueueExample {

    public static void main(String[] args) {

        Queue<Integer> queue = new LinkedList<>();

        queue.add(10);

        queue.add(20);

        queue.add(30);

        System.out.println(queue.poll()); // Output: 10 (FIFO)

        System.out.println(queue); // Output: [20, 30]

    }

}
```

Set

An unordered collection that does **not** allow duplicate elements. Popular implementations include **HashSet, LinkedHashSet, and TreeSet**.

Example:

```
public class SetExample {

    public static void main(String[] args) {

        Set<String> set = new HashSet<>();

        set.add("Apple");

        set.add("Banana");

        set.add("Apple"); // Duplicate ignored
        System.out.println(set); // Output: [Apple, Banana] (Order not guaranteed)

    }

}
```

5. Exception Handling

An **exception** is an event that disrupts the normal execution flow of a program. When an exception occurs, Java creates an **exception object** and passes it to the runtime system, which searches for appropriate handling mechanisms.

A. Checked Exceptions

Checked exceptions are verified by the compiler at compile-time. If a method has a checked exception, it must either handle it using a try-catch block or declare it using the throws keyword.

Examples of Checked Exceptions:

- IOException
- SQLException
- ClassNotFoundException

Example:

```
public class CheckedExceptionExample {

    public static void main(String[] args) {

        try {

            Scanner sc = new Scanner(new File("nonexistent.txt")); // May throw
FileNotFoundException

        } catch (FileNotFoundException e) {

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

        }

    }

}
```

B. Unchecked Exceptions

Unchecked exceptions (runtime exceptions) are **not checked** at compile-time and occur due to logical programming errors. These exceptions extend the RuntimeException class.

Examples of Unchecked Exceptions:

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException
- ClassCastException

Example:

```
public class UncheckedExceptionExample {

    public static void main(String[] args) {

        try {

            int result = 10 / 0; // Throws ArithmeticException

            System.out.println(result);

        } catch (ArithmeticException e) {

            System.out.println("Cannot divide by zero: " + e.getMessage());

        }

    }

}
```

C. Exception Handling Mechanisms

The try-catch Block

The try-catch block is used to handle exceptions. The code that might throw an exception is placed inside the try block, and the code to handle the exception is placed inside the catch block.

syntax:

```
try {

        // Code that might throw an exception

        // ...

} catch (ExceptionType e) {
        // Code to handle the exception

        // ...

}
```

Multiple catch Blocks

You can have multiple catch blocks to handle different types of exceptions that may occur within the try block.

syntax:

```
try {

        // Code that might throw an exception

        // ...

} catch (ExceptionType1 e1) {

        // Code to handle ExceptionType1

        // ...

} catch (ExceptionType2 e2) {

        / Code to handle ExceptionType2

        // ...

}
```

The finally Block

The finally block contains code that will be executed regardless of whether an exception is thrown or not. It is often used for cleanup operations, such as closing resources.

syntax:

```
try {

        // Code that might throw an exception

        // ...

} catch (ExceptionType e) {

        // Code to handle the exception

        // ...
} finally {
    // Code that will be executed regardless of exceptions
    // ...
}
```

The throw & throws Keyword

You can use the throw keyword to explicitly throw an exception in your code. This is useful when you want to signal an exceptional condition.

The throws clause is used in a method signature to declare that the method may throw checked exceptions. It informs the caller that the method might encounter certain exceptional conditions, and the caller is responsible for handling these exceptions.

Example:

```
public class ThrowThrowsExample {
  public static void main(String[] args) {
    try {
      performDivision(10, 0); // This will cause an exception
    } catch (ArithmeticException e) {
      System.out.println("Caught an exception: " + e.getMessage());
    }
  }
  // 'throws' indicates this method might throw an ArithmeticException
  public static int performDivision(int numerator, int denominator) throws
ArithmeticException {
    if (denominator == 0) {
      // 'throw' creates and throws an ArithmeticException
      throw new ArithmeticException("Cannot divide by zero!");
    }
    return numerator / denominator;
    }
}
```

D. Custom Exceptions

Java allows developers to create their own exceptions by extending the Exception or RuntimeException class.

Example:

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            checkEligibility(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
    static void checkEligibility(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older.");
        }
    }
}
```