Abstract Keyword Continued

Interfaces

While an `abstract class` can provide a partial implementation (concrete methods), an **interface** is a completely abstract type. It is a contract that specifies a set of abstract methods that a class *must* implement.

Key Characteristics of Interfaces:

- Declared using the `interface` keyword.
- **Cannot be instantiated.**
- Methods in an interface are implicitly `public` and `abstract` (you don't need to write the keywords).
- Variables in an interface are implicitly `public`, `static`, and `final` (they are constants).
- A class uses the `implements` keyword to use an interface.
- A class can **implement multiple interfaces**, which is how Java achieves a form of multiple inheritance.

Abstract Class vs. Interface

Feature	Abstract Class	Interface
Methods	Can have both abstract and concrete (non-abstract) methods.	Can only have abstract methods (before Java 8).
Variables	Can have instance variables (`final`, non-`final`, `static`, non-`static`).	Can only have `public static final` variables (constants).
Constructor	Has a constructor (called by subclasses).	Does not have a constructor.
Inheritance	A class can `extend` only one abstract class.	A class can `implement` multiple interfaces.
Keyword	`abstract`, `extends`	`interface`, `implements`
Purpose	To share common code among related subclasses (Is-A relationship).	To define a contract of behaviors for unrelated classes (Can-Do relationship).

Note: Since Java 8, interfaces can have `default` and `static` methods with implementations, which blurs the lines slightly, but the core purpose remains the same.

Example:

Let's model things that can be "drivable." A `Car` and a `Bicycle` are very different, but they both share the "drivable" behavior. An interface is perfect for this.

```java
// 1. The Interface: A contract for anything that is Drivable
interface Drivable {
    // Methods are public abstract by default
    void turn(String direction);
    void accelerate(int speed);
    void brake();
}

// 2. A class implementing the interface
class Car implements Drivable {
    @Override
    public void turn(String direction) {
        System.out.println("Car is turning " + direction);
    }

    @Override
    public void accelerate(int speed) {
        System.out.println("Car is accelerating to " + speed + " mph.");
    }

    @Override
    public void brake() {
        System.out.println("Car is braking.");
    }
}

// 3. A completely different class also implementing the same interface
class Bicycle implements Drivable {
    @Override
    public void turn(String direction) {
        System.out.println("Bicycle is turning " + direction + " by leaning.");
    }

    @Override
    public void accelerate(int speed) {
        System.out.println("Bicycle is accelerating to " + speed + " mph by pedaling
faster.");
    }

    @Override
    public void brake() {
        System.out.println("Bicycle is braking using hand brakes.");
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Drivable myCar = new Car();
        Drivable myBike = new Bicycle();

        myCar.accelerate(60);
        myCar.brake();

        System.out.println();

        myBike.accelerate(15);
        myBike.brake();
    }
}
```

What is Polymorphism

Polymorphism (from Greek, meaning "many forms") is the ability of an object, method, or variable to take on different forms. In OOP, it allows you to perform a single action in different ways.

There are two types of polymorphism in Java:

1. **Compile-time Polymorphism (Static Binding):** Achieved through **method overloading**. The compiler knows which method to call at compile time based on the method signature (name and parameter list).
2. **Runtime Polymorphism (Dynamic Binding):** Achieved through **method overriding**. An overridden method is resolved at runtime, not compile time. The actual object type (not the reference type) determines which method is executed. This is a core concept that relies on inheritance.

References and Objects

This is the key to understanding runtime polymorphism. In Java, a reference variable of a superclass or interface type can hold an object of any of its subclasses.

```
// Superclass reference holding a subclass object
Animal myPet = new Dog();

// Interface reference holding an implementing class object
Drivable vehicle = new Car();
```

When you call a method using this reference (`myPet.makeSound()`), the Java Virtual Machine (JVM) checks the *actual object's type* at runtime (`Dog`) and calls the overridden method from that class, not the method from the reference's class (`Animal`).

Method / Constructor Overloading

This is **compile-time polymorphism**. Overloading allows you to define multiple methods or constructors in the same class with the same name, as long as their **parameter lists are different**. The difference can be in the number of parameters, the type of parameters, or the order of parameters.

Example:

```
class Calculator {
    // Method Overloading
    public int add(int a, int b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }
    public double add(double a, double b) {
        return a + b;
    }
}

class User {
    String username;
    String password;
    String email;

    // Constructor Overloading
    public User(String username, String password) {
        this.username = username;
        this.password = password;
        this.email = "N/A";
    }

    public User(String username, String password, String email) {
        this.username = username;
        this.password = password;
        this.email = email;
    }
}
```

Super Keyword

The `super` keyword is a reference variable that is used to refer to the immediate parent class object.

It has two main uses:

1. **`super()`:** To call the constructor of the immediate parent class. This must be the *first statement* in a subclass constructor. If you don't explicitly call `super()`, the compiler will implicitly insert a call to the parent's no-argument constructor.
2. **`super.memberName`:** To access a method or variable of the parent class, especially when the subclass has overridden it.

Example (revisiting the `Dog` class):

```
class Animal {
    String name;

    public Animal(String name) {
        System.out.println("Animal constructor called.");
        this.name = name;
    }

    public void makeSound() {
        System.out.println("Some generic animal sound.");
    }
}

class Dog extends Animal {
    public Dog(String name) {
        super(name); // 1. Calls the Animal(String name) constructor. Must be the
first line.
        System.out.println("Dog constructor called.");
    }

    @Override
    public void makeSound() {
        super.makeSound(); // 2. Calls the makeSound() method from the Animal class.
        System.out.println("Woof! Woof!");
    }
}
```

Method / Constructor Overriding

This is **runtime polymorphism**. Overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

Rules for Method Overriding:

- The method name, parameter list, and return type must be the same (or a covariant return type).
- The access modifier in the overriding method cannot be more restrictive than the overridden method (e.g., you can't override a `public` method with a `private` one).
- `final` and `static` methods cannot be overridden.
- The `@Override` annotation is not mandatory but is highly recommended. It tells the compiler you intend to override a method, and it will generate an error if you fail to do so correctly (e.g., due to a typo).

Constructors cannot be overridden. This is because constructors are not inherited. A subclass has its own constructors, which are responsible for initializing the subclass's state (though they must call a superclass constructor).

Example of Runtime Polymorphism:

```
public class PolymorphismExample {
    public static void main(String[] args) {
        // Parent class reference holding a parent class object
        Animal myAnimal = new Animal("Generic Animal");

        // Parent class reference holding a child class object
        Animal myDog = new Dog("Buddy");

        // Parent class reference holding another child class object
        Animal myCat = new Cat("Whiskers"); // Assume Cat class exists and extends
Animal

        myAnimal.makeSound(); // Calls Animal's method
        myDog.makeSound();    // RUNTIME DECISION: Calls Dog's overridden method
        myCat.makeSound();    // RUNTIME DECISION: Calls Cat's overridden method
    }
}
// Assume Cat class:
```

```
// class Cat extends Animal { ... @Override public void makeSound() {
System.out.println("Meow!"); } ... }
```

Final Keyword Revisited

Let's connect the `final` keyword back to these principles:

- **`final` variable:** Creates a constant. Its value cannot be changed.
- **`final` method: Cannot be overridden** by a subclass. This is used when you want to guarantee that a method's implementation will not be changed down the inheritance chain.
- **`final` class: Cannot be extended** (inherited from). This is used for security and immutability. The `String` class, for example, is `final`.

Pass by Value vs. Pass by Reference

This is a very important and often misunderstood concept in Java.

The simple rule is: Java is always pass-by-value.

What does this mean? When you pass a variable to a method, a **copy** of that variable is made, and the method receives that copy.

1. **For Primitive Types (`int` , `double` , `boolean` , etc.):**
    - A copy of the *actual value* is passed.
    - Changes made to the parameter inside the method do **not** affect the original variable outside the method.
2. **For Reference Types (Objects, Arrays):**
    - A copy of the *reference (memory address)* is passed.
    - Both the original reference and the method's parameter now point to the **same object** in the heap.
    - If you use the parameter to change the *state of the object* (e.g., `myObject.setValue(10)`), the change will be visible outside the method because the original reference points to that same modified object.
    - However, if you try to reassign the parameter to a *new object* inside the method (`myObject = new MyObject()`), this only changes the method's local copy of the reference. It does **not** affect the original reference outside the method.

Example to Clarify:

```
class Balloon {
    String color;
    public Balloon(String color) { this.color = color; }
}

public class PassByValueExample {
    public static void main(String[] args) {
        // --- Primitive Example ---
        int originalValue = 10;
        System.out.println("Before method: " + originalValue);
        modifyPrimitive(originalValue);
        System.out.println("After method: " + originalValue); // Stays 10
        System.out.println();

        // --- Reference Example ---
        Balloon originalBalloon = new Balloon("Red");
        System.out.println("Before method: " + originalBalloon.color);
        modifyObjectState(originalBalloon);
        System.out.println("After method: " + originalBalloon.color); // Changes to
Blue

        System.out.println();

        System.out.println("Before method (reassign): " + originalBalloon.color);
        reassignObjectReference(originalBalloon);
        System.out.println("After method (reassign): " + originalBalloon.color); //
Stays Blue
    }

    public static void modifyPrimitive(int valueCopy) {
        valueCopy = 20; // Modifies only the copy
    }

    public static void modifyObjectState(Balloon balloonCopyRef) {
        // balloonCopyRef is a copy of the reference, but points to the SAME object
        balloonCopyRef.color = "Blue"; // This modifies the object itself
    }

    public static void reassignObjectReference(Balloon balloonCopyRef) {
        // This makes the local copy of the reference point to a NEW object
        balloonCopyRef = new Balloon("Green");
        // The original reference outside this method is unaffected.
    }
}
```

Exception & File Handling

What is an Exception

An **exception** is an unwanted or unexpected event that occurs during the execution of a program, disrupting its normal flow. When an error occurs within a method, the method creates an object (an exception object) and hands it off to the runtime system. This exception object contains information about the error, including its type and the state of the program when the error occurred. This process is called **throwing an exception**.

If the exception is not handled (or "caught"), the program will terminate abruptly, and the Java runtime will print a message to the console with a **stack trace**, which shows the sequence of method calls that led to the error.

Common examples of exceptions include:

- Trying to divide a number by zero (`ArithmeticException`).

- Trying to access an array element with an invalid index (`ArrayIndexOutOfBoundsException`).
- Trying to use a method on an object that is `null` (`NullPointerException`).
- Trying to open a file that doesn't exist (`FileNotFoundException`).

Try-Catch

The core mechanism for handling exceptions in Java is the `try-catch` block.

- **`try` block:** You place the code that might throw an exception inside the `try` block.
- **`catch` block:** This block immediately follows the `try` block. If an exception of a specific type occurs in the `try` block, the `try` block is abandoned, and the corresponding `catch` block is executed. A `catch` block is an exception handler that takes the type of exception it can handle as an argument.

Syntax:

```
try {
    // Code that might throw an exception
} catch (ExceptionType1 e1) {
    // Code to handle the ExceptionType1
} catch (ExceptionType2 e2) {
    // Code to handle the ExceptionType2
}
```

Example:

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            // This code might cause an ArithmeticException
            int result = 10 / 0;
            System.out.println("Result: " + result); // This line will not be reached
        } catch (ArithmeticException e) {
            // This block executes because an ArithmeticException was caught
            System.out.println("An error occurred: Cannot divide by zero.");
            // e.printStackTrace(); // A useful method to print the stack trace for
debugging
        }

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

        int[] numbers = {1, 2, 3};
        try {
            // This code might cause an ArrayIndexOutOfBoundsException
            System.out.println("Accessing element at index 5: " + numbers[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("An error occurred: Invalid array index.");
        }

        System.out.println("Program finished successfully.");
    }
}
```

Types of Exception

In Java, all exception and error types are subclasses of the `Throwable` class. The hierarchy is broadly divided into three categories:

1. **Checked Exceptions:**
    - These are exceptions that a well-written application should anticipate and recover from.
    - They are checked at **compile-time**. If a method can throw a checked exception, it must either handle it with a `try-catch` block or declare that it throws the exception using the `throws` keyword.
    - Examples: `IOException`, `FileNotFoundException`, `SQLException`.
2. **Unchecked Exceptions (Runtime Exceptions):**
    - These exceptions represent defects in the program logic and are generally not expected to be caught. They occur at **runtime**.
    - The compiler does not force you to handle them. They are subclasses of `RuntimeException`.
    - Examples: `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `IllegalArgumentException`.
3. **Errors:**
    - These are not exceptions but problems that arise beyond the control of the user or the programmer. They are typically ignored in your code because you can rarely do anything about them.
    - They are related to the runtime environment itself.
    - Examples: `OutOfMemoryError`, `StackOverflowError`.

Throw and Throws

- **`throw` keyword:**
    - Used to **manually throw an exception** from a method or any block of code.
    - You can throw either a newly instantiated exception (`throw new MyException();`) or an exception that you just caught.
    - It is used within a method's body.
- **`throws` keyword:**
    - Used in a **method signature** to declare that the method might throw one or more checked exceptions.
    - It delegates the responsibility of handling the exception to the caller method. The caller must then either handle it with `try-catch` or declare it with `throws` as well.

Example:

```java
public class ThrowThrowsExample {

    // This method DECLARES that it might throw an ArithmeticException
    public static void checkAge(int age) throws ArithmeticException {
        if (age < 18) {
            // This method MANUALLY THROWS an ArithmeticException
            throw new ArithmeticException("Access denied - You must be at least 18
years old.");
        } else {
            System.out.println("Access granted - You are old enough!");
        }
    }

    public static void main(String[] args) {
        // The calling method (main) must handle the declared exception
        try {
            checkAge(15); // This will throw the exception
        } catch (ArithmeticException e) {
            System.out.println("Caught an exception: " + e.getMessage());
        }

        try {
            checkAge(20); // This will not throw an exception
        } catch (ArithmeticException e) {
            System.out.println("This won't be printed.");
        }
    }
}
```

Finally Block

The **`finally` block** is an optional block that can follow a `try-catch` structure. The code inside the `finally` block is **always executed**, regardless of whether an exception was thrown or caught.

Its primary purpose is to execute cleanup code, such as closing files, releasing network connections, or closing database connections, to ensure that resources are not left open.

Execution Scenarios:

1. No exception occurs -> `try` -> `finally` -> rest of the program.
2. An exception occurs and is caught -> `try` -> `catch` -> `finally` -> rest of the program.
3. An exception occurs and is not caught -> `try` -> `finally` -> program terminates.

Example:

```
public class FinallyBlockExample {
    public static void main(String[] args) {
        try {
            System.out.println("Inside the try block.");
            // int result = 10 / 0; // Uncomment to see exception scenario
        } catch (ArithmeticException e) {
            System.out.println("Inside the catch block.");
        } finally {
            // This block will always execute.
            System.out.println("Inside the finally block. This is for cleanup.");
        }
```

```java
        System.out.println("Program continues...");
    }
}
```

Custom Exceptions

You can create your own exception classes by extending one of the existing exception classes, usually `Exception` (for a checked exception) or `RuntimeException` (for an unchecked exception).

Creating custom exceptions makes your code more readable and allows you to create specific exception types for your application's business logic.

Example:

```java
// 1. Create a custom exception class
class InsufficientFundsException extends Exception {
    // Constructor that takes a message
    public InsufficientFundsException(String message) {
        super(message); // Pass the message to the parent Exception class
    }
}

// 2. A class that uses the custom exception
class BankAccount {
    private double balance;

    public BankAccount(double balance) {
        this.balance = balance;
    }

    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Withdrawal amount of $" + amount + " exceeds the current balance of $" + balance);
        }
        balance -= amount;
        System.out.println("Successfully withdrew $" + amount);
    }
}

// 3. Main class to test it
public class CustomExceptionExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(100.0);
        try {
            account.withdraw(50.0);  // This will work
            account.withdraw(60.0);  // This will throw the exception
        } catch (InsufficientFundsException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

FileWriter Class

The `java.io.FileWriter` class is used to write character data to a file. It's a simple way to write text files.

It's important to close the writer when you are done to ensure that all the data is written to the file and to release the system resources. The `try-with-resources` statement is the best way to do this.

Example:

```java
import java.io.FileWriter;
import java.io.IOException;

public class FileWriterExample {
    public static void main(String[] args) {
        // Using try-with-resources to automatically close the FileWriter
        try (FileWriter writer = new FileWriter("output.txt")) {
            writer.write("Hello, this is a line of text.\n");
            writer.write("This is another line.\n");
            writer.write("Java file handling is easy!");
            System.out.println("Successfully wrote to the file.");
        } catch (IOException e) {
            System.out.println("An error occurred while writing to the file.");
            e.printStackTrace();
        }
    }
}
```

After running this code, a file named `output.txt` will be created in your project directory with the specified text.

FileReader Class

The `java.io.FileReader` class is used to read character data from a file. You can read the file character by character.

For more efficient reading, especially for larger files, `FileReader` is often wrapped in a `BufferedReader`, which reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

Example:

```java
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class FileReaderExample {
    public static void main(String[] args) {
        // Using try-with-resources to automatically close FileReader and
BufferedReader
        try (FileReader fileReader = new FileReader("output.txt");
             BufferedReader bufferedReader = new BufferedReader(fileReader)) {
```

```
            System.out.println("Reading from the file:");
            String line;
            // Read the file line by line until the end is reached (readLine() returns
null)
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }

        } catch (IOException e) {
            System.out.println("An error occurred while reading the file.");
            e.printStackTrace();
        }
    }
}
```

This code will open the `output.txt` file created by the previous example and print its contents to the console.

Collections & Generics

Variable Arguments (Varargs)

Variable Arguments, or **Varargs**, allow a method to accept a variable number of arguments (zero or more) of the same type. It provides a shorthand for situations where you would otherwise need to create an array manually to pass multiple arguments.

Inside the method, the varargs parameter is treated as an array of its specified type.

Syntax: `returnType methodName(dataType... variableName)`

Rules:

- A method can have only one varargs parameter.
- The varargs parameter must be the last parameter in the method's signature.

Example:

```
public class VarargsExample {

    // This method can accept zero or more integer arguments.
    public static int sum(int... numbers) {
        System.out.println("Number of arguments: " + numbers.length);
        int total = 0;
        for (int num : numbers) {
            total += num;
        }
        return total;
    }

    public static void main(String[] args) {
        System.out.println("Sum is: " + sum());          // Zero arguments
        System.out.println("Sum is: " + sum(10));         // One argument
        System.out.println("Sum is: " + sum(10, 20, 30)); // Three arguments
        int[] myNumbers = {5, 10, 15, 20};
        System.out.println("Sum is: " + sum(myNumbers));  // Can also pass an array
    }
}
```

Wrapper Classes & Autoboxing

Java has two categories of data types: primitive types (`int`, `char`, `double`, etc.) and reference types (objects). The **Collections Framework** can only store objects, not primitive types.

Wrapper Classes are a set of classes in `java.lang` that "wrap" a primitive data type into an object. Each primitive type has a corresponding wrapper class.

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

Autoboxing and Unboxing:

- **Autoboxing:** The automatic conversion that the Java compiler makes between a primitive type and its corresponding wrapper class.
- **Unboxing:** The reverse process of converting an object of a wrapper type back to its corresponding primitive value.

This process makes the code cleaner because you don't have to explicitly convert between primitives and objects when using collections.

Example:

```
import java.util.ArrayList;
import java.util.List;

public class AutoboxingExample {
    public static void main(String[] args) {
        // Before Java 5 (Manual boxing)
        // Integer integerObject = new Integer(10);

        // Autoboxing (primitive int is automatically converted to an Integer object)
        Integer autoBoxed = 100;

        // Unboxing (Integer object is automatically converted to a primitive int)
        int unBoxed = autoBoxed;

        System.out.println("Autoboxed value: " + autoBoxed);
        System.out.println("Unboxed value: " + unBoxed);

        // Autoboxing in collections
        List<Integer> numberList = new ArrayList<>();
        numberList.add(1); // Autoboxing: int 1 is converted to new Integer(1)
        numberList.add(2);

        int firstNum = numberList.get(0); // Unboxing: Integer object is converted
back to int
        System.out.println("First number from list: " + firstNum);
    }
}
```

Collections Library

The **Java Collections Framework** is a unified architecture for representing and manipulating collections of objects. It provides a set of interfaces and classes to help developers manage data more efficiently.

The core of the framework consists of several key interfaces:

- **`Collection`**: The root interface of the hierarchy.
- **`List`**: An ordered collection that allows duplicate elements.
- **`Set`**: A collection that does not allow duplicate elements.
- **`Queue`**: A collection used to hold elements prior to processing, typically in a FIFO (First-In, First-Out) order.
- **`Map`**: An object that maps keys to values. It is not a true `Collection` (it doesn't extend the `Collection` interface) but is considered part of the framework.

List Interface

A `List` is an ordered collection (also known as a sequence) that allows duplicate elements. Elements can be accessed by their integer index (position).

Common Implementations:

- **`ArrayList`**: Implemented as a resizable array. Provides fast random access (getting an element by index) but is slower for insertions and deletions in the middle of the list.
- **`LinkedList`**: Implemented as a doubly-linked list. Faster for insertions and deletions, but slower for random access. Also implements the `Queue` interface.

Example:

```
import java.util.ArrayList;
import java.util.List;

public class ListInterfaceExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        // Adding elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Apple"); // Duplicates are allowed

        System.out.println("List of fruits: " + fruits);

        // Accessing an element by index
        System.out.println("Element at index 1: " + fruits.get(1));

        // Getting the size of the list
        System.out.println("Size of the list: " + fruits.size());

        // Removing an element
        fruits.remove("Banana");
        System.out.println("List after removing Banana: " + fruits);

        // Iterating over the list
        System.out.println("Iterating through the list:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

Queue Interface

A `Queue` is a collection designed for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, extraction, and inspection operations. Queues typically, but do not necessarily, order elements in a FIFO (First-In, First-Out) manner.

Common Implementations:

- **`LinkedList`**: A common general-purpose implementation.
- **`PriorityQueue`**: Orders elements based on their natural ordering or a supplied `Comparator`.

Key Methods: There are two sets of methods for core operations: one that throws an exception if the operation fails, and another that returns a special value (`null` or `false`).

Operation	Throws Exception	Returns Special Value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Example:

```
import java.util.LinkedList;
import java.util.Queue;
```

```java
public class QueueInterfaceExample {
    public static void main(String[] args) {
        Queue<String> customerLine = new LinkedList<>();

        // offer() adds elements to the back of the queue
        customerLine.offer("Alice");
        customerLine.offer("Bob");
        customerLine.offer("Charlie");

        System.out.println("Current line: " + customerLine);

        // peek() examines the head of the queue without removing it
        System.out.println("Next in line: " + customerLine.peek());

        // poll() removes and returns the head of the queue
        String servedCustomer = customerLine.poll();
        System.out.println("Serving: " + servedCustomer);
        System.out.println("Line after serving: " + customerLine);

        servedCustomer = customerLine.poll();
        System.out.println("Serving: " + servedCustomer);
        System.out.println("Line after serving: " + customerLine);
    }
}
```

Set Interface

A `Set` is a collection that contains **no duplicate elements**. It models the mathematical set abstraction.

Common Implementations:

- **`HashSet`**: Stores elements in a hash table. It is the best-performing implementation but makes no guarantees concerning the iteration order.
- **`LinkedHashSet`**: A hash table and linked list implementation that maintains the insertion order of elements.
- **`TreeSet`**: Stores elements in a sorted order (e.g., alphabetical, numerical).

Example:

```java
import java.util.HashSet;
import java.util.Set;

public class SetInterfaceExample {
    public static void main(String[] args) {
        Set<String> uniqueColors = new HashSet<>();

        uniqueColors.add("Red");
        uniqueColors.add("Green");
        uniqueColors.add("Blue");

        // Trying to add a duplicate element
        boolean isAdded = uniqueColors.add("Red");
        System.out.println("Was 'Red' added again? " + isAdded); // false
        System.out.println("Set of unique colors: " + uniqueColors); // Order is not
guaranteed

        // Check for an element
        System.out.println("Does the set contain 'Green'? " +
uniqueColors.contains("Green"));

        // Remove an element
        uniqueColors.remove("Blue");
        System.out.println("Set after removing Blue: " + uniqueColors);
    }
}
```

Collections Class

The `java.util.Collections` class (note the 's') is a utility class that consists exclusively of `static` methods that operate on or return collections. It provides useful functionality for manipulating collections.

Useful Methods:

- `sort(List<T> list)` : Sorts the elements of a list.
- `reverse(List<?> list)` : Reverses the order of elements in a list.
- `shuffle(List<?> list)` : Randomly shuffles the elements in a list.
- `max(Collection<? extends T> coll)` : Returns the maximum element in a collection.
- `min(Collection<? extends T> coll)` : Returns the minimum element.
- `frequency(Collection<?> c, Object o)` : Counts the occurrences of an object in a collection.

Example:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CollectionsClassExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(50);
        numbers.add(10);
        numbers.add(80);
        numbers.add(20);

        System.out.println("Original list: " + numbers);

        Collections.sort(numbers);
        System.out.println("Sorted list: " + numbers);

        Collections.reverse(numbers);
        System.out.println("Reversed list: " + numbers);

        Collections.shuffle(numbers);
        System.out.println("Shuffled list: " + numbers);

        System.out.println("Max value in list: " + Collections.max(numbers));
    }
}
```

Map Interface

A `Map` is an object that maps **keys** to **values**. A map cannot contain duplicate keys; each key can map to at most one value. It models the function abstraction.

Common Implementations:

- **`HashMap`**: The most commonly used implementation. It makes no guarantees about the order of the map.
- **`LinkedHashMap`**: Maintains the insertion order of keys.
- **`TreeMap`**: Keeps the keys in sorted order.

Example:

```
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.TreeMap;

public class MapDemonstration {

    public static void main(String[] args) {

        // --- 1. Creating a Map (using HashMap as an example) ---
        // A Map to store student IDs (Integer) and their names (String)
        Map<Integer, String> studentNames = new HashMap<>();
        System.out.println("1. Initial HashMap: " + studentNames); // {}

        // --- 2. put(K key, V value): Adds an entry to the map ---
        studentNames.put(101, "Alice");
        studentNames.put(102, "Bob");
        studentNames.put(103, "Charlie");
        System.out.println("2. After adding entries: " + studentNames); // {101=Alice,
102=Bob, 103=Charlie}

        // --- 3. get(Object key): Retrieves the value associated with a key ---
        String nameOf102 = studentNames.get(102);
        System.out.println("3. Name of student 102: " + nameOf102); // Bob

        String nameOf200 = studentNames.get(200); // Key not present
        System.out.println("3. Name of student 200 (not found): " + nameOf200); //
null

        // --- 4. containsKey(Object key): Checks if a key exists in the map ---
        boolean has101 = studentNames.containsKey(101);
        boolean has105 = studentNames.containsKey(105);
        System.out.println("4. Map contains key 101? " + has101); // true
        System.out.println("4. Map contains key 105? " + has105); // false
        // --- 5. containsValue(Object value): Checks if a value exists in the map ---
        boolean hasAlice = studentNames.containsValue("Alice");
        boolean hasDavid = studentNames.containsValue("David");
        System.out.println("5. Map contains value \"Alice\"? " + hasAlice); // true
        System.out.println("5. Map contains value \"David\"? " + hasDavid); // false

        // --- 6. replace(K key, V value): Replaces the value for an existing key ---
        studentNames.replace(101, "Alicia");
        System.out.println("6. After replacing 101: " + studentNames); // {101=Alicia,
102=Bob, 103=Charlie}

        // --- 7. putIfAbsent(K key, V value): Adds if key not already present ---
        studentNames.putIfAbsent(104, "David"); // 104 is new, so it's added
        studentNames.putIfAbsent(102, "Robert"); // 102 exists, so "Bob" remains
        System.out.println("7. After putIfAbsent: " + studentNames); // {101=Alicia,
102=Bob, 103=Charlie, 104=David}

        // --- 8. remove(Object key): Removes the entry for a specified key ---
        studentNames.remove(103);
        System.out.println("8. After removing 103: " + studentNames); // {101=Alicia,
102=Bob, 104=David}

        // --- 9. size(): Returns the number of key-value mappings in this map ---
        System.out.println("9. Current size of map: " + studentNames.size()); // 3

        // --- 10. isEmpty(): Returns true if this map contains no key-value mappings
---
        System.out.println("10. Is map empty? " + studentNames.isEmpty()); // false

        // --- 11. keySet(): Returns a Set view of the keys contained in this map ---
        System.out.println("11. All keys: " + studentNames.keySet()); // [101, 102,
104] (order may vary for HashMap)

        // --- 12. values(): Returns a Collection view of the values contained in this
map ---
        System.out.println("12. All values: " + studentNames.values()); // [Alicia,
Bob, David] (order may vary for HashMap)

        // --- 13. entrySet(): Returns a Set view of the mappings contained in this
map ---
        System.out.println("13. All entries: " + studentNames.entrySet()); //
[101=Alicia, 102=Bob, 104=David] (order may vary for HashMap)

        // --- 14. Iterating over a Map ---
        System.out.println("\n14. Iterating over the map:");

        // Option A: Using entrySet() to get both key and value
        System.out.println("   - Using entrySet():");
        for (Map.Entry<Integer, String> entry : studentNames.entrySet()) {
            System.out.println("     Student ID: " + entry.getKey() + ", Name: " +
entry.getValue());
```

```
        }

        // Option B: Using keySet() to get keys, then get() for values
        System.out.println("   - Using keySet():");
        for (Integer id : studentNames.keySet()) {
            System.out.println("     Student ID: " + id + ", Name: " +
studentNames.get(id));
        }

        // Option C: Using forEach (Java 8+)
        System.out.println("   - Using forEach (Java 8+):");
        studentNames.forEach((id, name) -> System.out.println("     Student ID: " + id
+ ", Name: " + name));

        // --- 15. clear(): Removes all of the mappings from this map ---
        studentNames.clear();
        System.out.println("15. After clear: " + studentNames); // {}
        System.out.println("15. Is map empty after clear? " + studentNames.isEmpty());
// true


        System.out.println("\n--- Demonstrating different Map implementations ---");

        // --- LinkedHashMap example (maintains insertion order) ---
        Map<String, Double> stockPrices = new LinkedHashMap<>();
        stockPrices.put("GOOG", 1500.00);
        stockPrices.put("MSFT", 250.75);
        stockPrices.put("AMZN", 3300.50);
        System.out.println("\nLinkedHashMap (insertion order maintained):");
        for (Map.Entry<String, Double> entry : stockPrices.entrySet()) {
            System.out.println("  " + entry.getKey() + " : " + entry.getValue());
        }
        // Expected output order: GOOG, MSFT, AMZN

        // --- TreeMap example (sorted by natural order of keys) ---
        Map<String, Integer> wordCounts = new TreeMap<>();
        wordCounts.put("apple", 5);
        wordCounts.put("zebra", 2);
        wordCounts.put("banana", 8);
        wordCounts.put("cat", 3);
        System.out.println("\nTreeMap (keys sorted alphabetically):");
        for (Map.Entry<String, Integer> entry : wordCounts.entrySet()) {
            System.out.println("  " + entry.getKey() + " : " + entry.getValue());
        }
        // Expected output order: apple, banana, cat, zebra
    }
}
```

Enums

An **enum (enumeration)** is a special "class" that represents a group of constants. Using enums can make your code more readable and less prone to errors compared to using integer or string constants.

Example:

```java
// Defining an enum for difficulty levels
enum Level {
    EASY,
    MEDIUM,
    HARD
}

public class EnumExample {
    public static void main(String[] args) {
        Level myLevel = Level.MEDIUM;

        switch (myLevel) {
            case EASY:
                System.out.println("The level is easy.");
                break;
            case MEDIUM:
                System.out.println("The level is medium.");
                break;
            case HARD:
                System.out.println("The level is hard.");
                break;
        }

        // Iterating over all enum constants
        System.out.println("\nAll available levels:");
        for (Level lvl : Level.values()) {
            System.out.println(lvl);
        }
    }
}
```

1. Iterator

An **Iterator** is an interface in Java that provides a standard way to traverse through the elements of a collection, one by one. It is considered a "universal Java cursor" because it can be used with all collection classes like `ArrayList`, `HashSet`, and `LinkedList`.

Key Methods of Iterator:

- `boolean hasNext()`: Returns `true` if the iteration has more elements.
- `E next()`: Returns the next element in the iteration and advances the cursor. It throws a `NoSuchElementException` if there are no more elements.
- `void remove()`: Removes the last element returned by the `next()` method from the underlying collection. This is the only safe way to modify a collection while iterating over it.

Example: Using an Iterator

This example demonstrates how to iterate through an `ArrayList` of strings, print each element, and safely remove an element during the iteration.

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample {
    public static void main(String[] args) {
        // Create an ArrayList of strings
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Date");

        // Get an Iterator for the ArrayList
        Iterator<String> iterator = fruits.iterator();

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

        // Loop through the collection using the iterator
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            System.out.println("Processing: " + fruit);

            // Safely remove the element "Cherry"
            if ("Cherry".equals(fruit)) {
                iterator.remove();
                System.out.println(">> Removed " + fruit);
            }
        }

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

Output:

```
Fruits before removal: [Apple, Banana, Cherry, Date]
Processing: Apple
Processing: Banana
Processing: Cherry
>> Removed Cherry
Processing: Date
Fruits after removal: [Apple, Banana, Date]
```

---

2. Comparable

The **Comparable** interface is used to define the "natural ordering" of a class. When a class implements `Comparable`, it gains a default sort order. This interface is located in the `java.lang` package and requires the implementation of a single method: `compareTo()`.

Key Method of Comparable:

- `int compareTo(T obj)`: Compares the current object with the specified object (`obj`). It should return:
  - A **negative integer** if the current object is less than `obj`.
  - **Zero** if the current object is equal to `obj`.
  - A **positive integer** if the current object is greater than `obj`.

Example: Implementing Comparable

In this example, a `Student` class implements `Comparable` to define a natural sort order based on the student's `id`.

```java
import java.util.ArrayList;
import java.util.Collections;

class Student implements Comparable<Student> {
    private int id;
    private String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Student{" + "id=" + id + ", name='" + name + '\'' + '}';
    }

    // Implement the compareTo method for natural ordering (by id)
    @Override
    public int compareTo(Student other) {
        // this.id < other.id  -> negative
        // this.id == other.id -> 0
        // this.id > other.id  -> positive
        return this.id - other.id;
    }
}

public class ComparableExample {
    public static void main(String[] args) {
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(101, "Alice"));
        students.add(new Student(103, "Charlie"));
        students.add(new Student(102, "Bob"));
        System.out.println("Students before sorting: " + students);

        // Sort the list using the natural order defined in the Student class
        Collections.sort(students);

        System.out.println("Students after sorting by ID: " + students);
    }
}
```

Output:

```
Students before sorting: [Student{id=101, name='Alice'}, Student{id=103,
name='Charlie'}, Student{id=102, name='Bob'}]
Students after sorting by ID: [Student{id=101, name='Alice'}, Student{id=102,
name='Bob'}, Student{id=103, name='Charlie'}]
```

---

3. Comparator

The **Comparator** interface is used when you need to define custom or multiple sorting strategies for a class. Unlike `Comparable`, you don't need to modify the class whose objects you want to sort. Instead, you create a separate class that implements the `Comparator` interface. This interface is located in the `java.util` package.

Key Method of Comparator:

- `int compare(T o1, T o2)`: Compares its two arguments for order. It should return:
  - A **negative integer** if the first argument (`o1`) is less than the second (`o2`).
  - **Zero** if the arguments are equal.
  - A **positive integer** if the first argument is greater than the second.

Example: Implementing Comparator

Using the same `Student` class from the previous example, we can create different `Comparator`s to sort the list by name or in descending order of ID.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

// Sorts Students by their name alphabetically
class SortByName implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.getName().compareTo(s2.getName());
    }
}

// Sorts Students by their ID in descending order
class SortByIdDesc implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s2.getId() - s1.getId();
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student(101, "Alice"));
        students.add(new Student(103, "Charlie"));
        students.add(new Student(102, "Bob"));

        System.out.println("Original list: " + students);

        // Sort by name using the SortByName comparator
        Collections.sort(students, new SortByName());
        System.out.println("Sorted by name: " + students);

        // Sort by ID descending using the SortByIdDesc comparator
        Collections.sort(students, new SortByIdDesc());
        System.out.println("Sorted by ID (descending): " + students);
    }
}
```

Output:

```
Original list: [Student{id=101, name='Alice'}, Student{id=103, name='Charlie'},
Student{id=102, name='Bob'}]
Sorted by name: [Student{id=101, name='Alice'}, Student{id=102, name='Bob'},
Student{id=103, name='Charlie'}]
Sorted by ID (descending): [Student{id=103, name='Charlie'}, Student{id=102,
name='Bob'}, Student{id=101, name='Alice'}]
```

Comparable vs. Comparator: Key Differences

Feature	Comparable	Comparator
Package	`java.lang`	`java.util`
Method	`compareTo(T obj)`	`compare(T o1, T o2)`
Implementation	Implemented by the class itself whose instances are to be sorted.	Implemented in a separate class.
Use Case	Defines a single, natural sorting order for a class.	Defines multiple, external, or custom sorting orders.
Modification	Requires modification of the source code of the class.	Does not require any change in the class being sorted.
Sorting Call	`Collections.sort(list);`	`Collections.sort(list, new MyComparator());`

Generics & Diamond Operators

Generics add a layer of abstraction over types. They allow you to define classes, interfaces, and methods where the type of data they operate on is specified as a parameter.

Benefits:

- **Stronger type checks at compile time:** You can catch invalid type errors during compilation rather than at runtime.
- **Elimination of casts:** You don't need to manually cast objects retrieved from a collection.
- **Enabling programmers to implement generic algorithms:** You can write code that works with different types of collections while remaining type-safe.

The **Diamond Operator (`<>`)** was introduced in Java 7 as a convenience. It allows the compiler to infer the generic type from the variable declaration, reducing boilerplate code.

Example:

```java
import java.util.ArrayList;
import java.util.List;

public class GenericsExample {
    public static void main(String[] args) {
        // Without Generics (the old way)
        List oldList = new ArrayList();
        oldList.add("hello");
        oldList.add(123); // No compile-time error, but this can cause runtime errors
        // String s = (String) oldList.get(1); // Throws ClassCastException at runtime

        // With Generics (the modern, safe way)
        // Before Java 7:
        // List<String> stringList = new ArrayList<String>();

        // With Diamond Operator (Java 7+):
        List<String> stringList = new ArrayList<>(); // The compiler infers <String>

        stringList.add("Apple");
        stringList.add("Banana");
        // stringList.add(10); // COMPILE-TIME ERROR! Prevents adding the wrong type.

        // No cast is needed
        String fruit = stringList.get(0);
        System.out.println("First fruit: " + fruit.toUpperCase());
    }
}
```

Multithreading & Executor Service

Intro to Multi-threading

A **process** is an instance of a program running on a computer (e.g., your web browser, your IDE). Each process has its own memory space.

A **thread** is the smallest unit of execution within a process. A single process can have multiple threads, each executing a different part of the program's code. This is called **multithreading**.

All threads within a single process share the same memory space, which makes communication between them easy but also introduces challenges related to data consistency and safety (race conditions).

Why use multithreading?

- **Responsiveness:** In a user interface application, one thread can handle user input while another performs a long-running task in the background, preventing the UI from freezing.
- **Performance:** On multi-core processors, you can execute multiple threads in parallel, significantly speeding up CPU-intensive tasks.
- **Resource Utilization:** Threads can keep the CPU busy while other parts of the program are waiting for I/O operations (like reading a file or making a network request).

Creating a Thread

There are two primary ways to create a thread in Java:

1. **By Extending the `Thread` Class:**
    - Create a new class that `extends java.lang.Thread`.
    - Override the `run()` method. This method contains the code that the thread will execute.
    - Create an instance of your new class and call its `start()` method. The `start()` method initializes the new thread and calls its `run()` method.
2. **By Implementing the `Runnable` Interface:**
    - Create a new class that `implements java.lang.Runnable`.
    - Implement the `run()` method.
    - Create an instance of your `Runnable` class.
    - Create a new `Thread` object, passing your `Runnable` object to its constructor.
    - Call the `start()` method on the `Thread` object.

Implementing `Runnable` is generally preferred because it allows your class to extend another class (Java does not support multiple class inheritance), promoting better object-oriented design.

Example:

```java
// Method 1: Extending Thread
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("MyThread executing: " + i);
            try { Thread.sleep(500); } catch (InterruptedException e) {}
        }
    }
}

// Method 2: Implementing Runnable (Preferred)
```

```java
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("MyRunnable executing: " + i);
            try { Thread.sleep(500); } catch (InterruptedException e) {}
        }
    }
}

public class CreatingThreadExample {
    public static void main(String[] args) {
        // Create and start the thread from the extended class
        MyThread thread1 = new MyThread();
        thread1.start(); // Never call run() directly!

        // Create and start the thread from the implemented class
        MyRunnable myRunnable = new MyRunnable();
        Thread thread2 = new Thread(myRunnable);
        thread2.start();

        System.out.println("Main thread finished.");
    }
}
```

States of a Thread

A thread goes through several states in its lifecycle:

1. **NEW:** The thread has been created but has not yet been started (i.e., `start()` has not been called).
2. **RUNNABLE:** The thread is either currently running or is ready to run and waiting for its turn from the thread scheduler.
3. **BLOCKED:** The thread is waiting to acquire a monitor lock to enter a `synchronized` block/method.
4. **WAITING:** The thread is in an indefinite waiting state. It is waiting for another thread to perform a particular action (e.g., calling `notify()` or `notifyAll()`). This happens when `Object.wait()`, `Thread.join()`, or `LockSupport.park()` are called.
5. **TIMED_WAITING:** The thread is waiting for a specified period. This state is entered by calling `Thread.sleep()`, `Object.wait(timeout)`, or `Thread.join(timeout)`.
6. **TERMINATED:** The thread has completed its execution (its `run()` method has finished).

Thread Priority

You can suggest a priority for a thread to the thread scheduler using `setPriority()`. The scheduler *may* use this as a hint to give more CPU time to higher-priority threads. However, this is platform-dependent and not guaranteed.

- Priorities are integers from `1` (`Thread.MIN_PRIORITY`) to `10` (`Thread.MAX_PRIORITY`).
- The default priority is `5` (`Thread.NORM_PRIORITY`).

Join Method

The `join()` method allows one thread to wait for the completion of another. When you call `t.join()` from the currently executing thread, it will pause its own execution until thread `t` has finished (i.e., its state becomes `TERMINATED`).

This is useful when the main thread needs results from a worker thread before it can proceed.

Example:

```java
public class JoinMethodExample {
    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            System.out.println("Worker thread started. Doing some work for 3 seconds...");
            try { Thread.sleep(3000); } catch (InterruptedException e) {}
            System.out.println("Worker thread finished.");
        });

        worker.start();
        System.out.println("Main thread is waiting for the worker to finish.");

        // The main thread will pause here and wait for the 'worker' thread to terminate.
        worker.join();

        System.out.println("Main thread continues now that the worker is done.");
    }
}
```

Synchronize Keyword

When multiple threads share and modify the same data, you can run into problems like **race conditions**, where the final outcome depends on the unpredictable timing of thread execution.

The `synchronized` keyword provides a simple strategy for preventing thread interference and memory consistency errors. It acts as a **lock** (or monitor).

- **Synchronized Method:** When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object. No other thread can enter *any* synchronized method on the *same object* until the lock is released (which happens when the method returns).
- **Synchronized Block:** Allows you to lock a smaller section of code rather than the entire method, which can improve concurrency. You specify the object on which to lock.

Example:

```java
class Counter {
    private int count = 0;

    // This method is not thread-safe.
    // public void increment() { count++; }
    // This is a synchronized method. Only one thread can execute it at a time on a
given Counter instance.
    public synchronized void increment() {
        count++;
    }

    public int getCount() { return count; }
}

public class SynchronizedExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        // With synchronized, the result will always be 2000.
        // Without it, the result would be unpredictable and likely less than 2000 due
to race conditions.
        System.out.println("Final count: " + counter.getCount());
    }
}
```

Thread Communication

Java provides a mechanism for threads to communicate with each other using the `wait()`, `notify()`, and `notifyAll()` methods. These methods are defined in the `Object` class.

- **`wait()`**: Causes the current thread to release the lock and enter a `WAITING` state until another thread invokes `notify()` or `notifyAll()` on the same object.
- **`notify()`**: Wakes up a single thread that is waiting on this object's monitor.
- **`notifyAll()`**: Wakes up all threads that are waiting on this object's monitor.

Important: These methods **must** be called from within a `synchronized` block or method on the object being used as the lock.

Intro to Executor Service

Manually creating and managing threads (`new Thread()`) can be complex and inefficient. For every task, a new thread is created and then destroyed, which has overhead.

The **Executor Framework** (introduced in Java 5) abstracts away the details of thread management. The central component is the `ExecutorService` interface. You submit tasks (`Runnable` or `Callable`) to the service, and it handles executing them, often using a pool of reusable threads (**thread pool**).

Benefits:

- **Improved Performance:** Reusing existing threads avoids the overhead of thread creation.
- **Resource Management:** You can control the maximum number of concurrent threads, preventing your application from being overwhelmed.
- **Simplified Code:** You focus on the tasks to be done, not the mechanics of thread management.

Multiple Threads with Executor

The `Executors` utility class provides factory methods for creating common types of `ExecutorService` .

- `newFixedThreadPool(int nThreads)` : Creates a thread pool with a fixed number of threads.
- `newCachedThreadPool()` : Creates a thread pool that creates new threads as needed but will reuse previously constructed threads when they are available.
- `newSingleThreadExecutor()` : Creates an executor that uses a single worker thread.

Example:

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceExample {
    public static void main(String[] args) {
        // Create a thread pool with 2 threads.
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Submit 5 tasks to the executor. Since the pool size is 2,
        // only two tasks will run at a time.
        for (int i = 1; i <= 5; i++) {
            final int taskId = i;
            Runnable task = () -> {
                System.out.println("Task " + taskId + " started by thread: " +
Thread.currentThread().getName());
                try { Thread.sleep(2000); } catch (InterruptedException e) {}
                System.out.println("Task " + taskId + " finished.");
            };
            executor.submit(task);
        }

        // It's crucial to shut down the executor when you're done with it.
        // shutdown() will allow currently running tasks to finish but won't accept
new tasks.
```

```
        executor.shutdown();

        System.out.println("All tasks submitted.");
    }
}
```

Returning Futures

The `Runnable` interface's `run()` method doesn't return a value. What if your task needs to compute a result and return it? For this, you use the `Callable` interface and `Future`.

- **`Callable<V>` Interface:** Similar to `Runnable`, but its `call()` method can return a value of type `V` and can throw a checked exception.
- **`Future<V>` Interface:** Represents the result of an asynchronous computation. When you submit a `Callable` to an `ExecutorService`, it returns a `Future` object immediately. The `Future` acts as a placeholder for the result, which may not be available yet. You can use its `get()` method to retrieve the result once it's complete. `get()` is a blocking call; it will wait until the result is ready.

Example:

```
import java.util.concurrent.*;

public class FutureExample {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        // Callable task that returns a String after a delay
        Callable<String> task = () -> {
            System.out.println("Task started in the background...");
            Thread.sleep(3000); // Simulate a long computation
            return "Hello from the future!";
        };

        System.out.println("Submitting the callable task...");
        Future<String> future = executor.submit(task);

        // You can do other work here while the task is running...
        System.out.println("Main thread is doing other work...");

        // Now, get the result from the Future. This will block until the task is
complete.
        System.out.println("Waiting for the result...");
        String result = future.get(); // This line blocks

        System.out.println("The result is: " + result);

        executor.shutdown();
    }
}
```

Functional Programming

What is Functional Programming

Functional Programming is a programming paradigm where programs are constructed by applying and composing **functions**. It treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

Key principles include:

- **Declarative Style (What, not How):** You describe *what* you want to do, not the step-by-step instructions on *how* to do it. Instead of writing a `for` loop to iterate and filter a list, you declare that you want a "filtered list."
- **Functions as First-Class Citizens:** Functions can be treated like any other variable. You can pass them as arguments to other functions, return them from functions, and store them in variables.
- **Immutability:** Data is preferably immutable. Instead of modifying an existing data structure, you create a new one with the updated values. This helps prevent side effects.
- **Avoiding Side Effects:** A function's output should depend only on its input arguments, without modifying any external state. This makes functions predictable and easier to reason about.

Lambda Expression

A **Lambda Expression** is an anonymous (unnamed) function that provides a concise way to implement a method from a functional interface. It allows you to treat functionality as a method argument, or code as data.

Syntax: `(parameter1, parameter2, ...) -> { code block }`

- If there is only one parameter, the parentheses `()` are optional.
- If the code block has only one statement, the curly braces `{}` and the `return` keyword are optional.

Example:

```java
import java.util.Collections;
import java.util.List;
import java.util.Arrays;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Charlie", "Alice", "Bob");

        // Pre-Java 8 way (using an anonymous inner class)
        Collections.sort(names, new java.util.Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                return a.compareTo(b);
            }
        });

        // With a Lambda Expression (much more concise)
        Collections.sort(names, (String a, String b) -> a.compareTo(b));
        // The compiler can infer the types, so it's even shorter
        Collections.sort(names, (a, b) -> a.compareTo(b));

        System.out.println(names); // [Alice, Bob, Charlie]
    }
}
```

What is a Stream

A **Stream** is not a data structure; it's a sequence of elements from a source that supports aggregate operations. Think of it as a conveyor belt where items (data elements) pass through a series of processing stations (operations).

Key Characteristics:

- **Source:** A stream gets its data from a source, such as a Collection, an Array, or an I/O channel.
- **Pipelining:** Stream operations are chained together to form a pipeline.
- **Laziness:** Intermediate operations are not executed until a terminal operation is invoked. This allows for significant optimizations.
- **Not a Data Store:** A stream doesn't store its elements. It carries values from a source through a pipeline.
- **Consumable:** A stream can be traversed only once.

Functional Interfaces

A **Functional Interface** is an interface that contains **exactly one abstract method**. It can have multiple `default` or `static` methods, but only one abstract method.

The `@FunctionalInterface` annotation is optional but recommended. It tells the compiler to produce an error if the interface doesn't meet the requirements.

Lambda expressions are instances of functional interfaces. The Java API provides many built-in functional interfaces in the `java.util.function` package:

- **`Predicate<T>`**: Takes an argument and returns a `boolean`. (Used for filtering).
- **`Consumer<T>`**: Takes an argument and returns nothing (`void`). (Used for `forEach`).
- **`Function<T, R>`**: Takes an argument of type `T` and returns a result of type `R`. (Used for `map`).
- **`Supplier<T>`**: Takes no arguments and returns a result.

Intermediate vs. Terminal Operations

A stream pipeline consists of a source, zero or more **intermediate operations**, and one **terminal operation**.

1. **Intermediate Operations:**
    - These operations are **lazy**. They don't perform any processing until a terminal operation is called.
    - They return a **new stream**, allowing operations to be chained.
    - Examples: `filter()`, `map()`, `sorted()`, `distinct()`.
2. **Terminal Operations:**

    - These operations **trigger the execution** of the stream pipeline.
    - They produce a result or a side-effect.
    - After the terminal operation is performed, the stream is considered **consumed** and cannot be reused.
    - Examples: `forEach()`, `collect()`, `reduce()`, `count()`, `max()`, `min()`.

Filtering & Reducing

These are two of the most common patterns used with streams.

- **Filtering (`filter(Predicate<T>)`):** An intermediate operation that takes a `Predicate` (a function that returns `true` or `false`) and produces a new stream containing only the elements that match the predicate.

- **Reducing (`reduce(...)`):** A terminal operation that combines all elements of a stream into a single result. It repeatedly applies a binary operator to the elements.

Example:

```java
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class FilterReduceExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Filter for even numbers, then find their sum using reduce.
        // The reduce operation takes an identity (starting value) and a lambda for
accumulation.
        int sumOfEvens = numbers.stream()
                                .filter(n -> n % 2 == 0) // Intermediate: filter for
evens
                                .reduce(0, (a, b) -> a + b); // Terminal: reduce to a
sum

        System.out.println("Sum of even numbers: " + sumOfEvens);

        List<String> words = Arrays.asList("hello", "functional", "world");
        Optional<String> longestWord = words.stream()
                                            .reduce((word1, word2) -> word1.length() >
word2.length() ? word1 : word2);

        longestWord.ifPresent(word -> System.out.println("The longest word is: " +
word));
    }
}
```

Method References

A **Method Reference** is a shorthand syntax for a lambda expression that executes just ONE method. It makes the code even more readable by referring to an existing method by name.

There are four kinds of method references:

Type	Syntax	Lambda Equivalent
Reference to a `static` method	`ClassName::methodName`	`(args) -> ClassName.methodName(args)`
Reference to an instance method of a particular object	`objectRef::methodName`	`(args) -> objectRef.methodName(args)`
Reference to an instance method of an arbitrary object of a particular type	`ClassName::methodName`	`(obj, args) -> obj.methodName(args)`
Reference to a constructor	`ClassName::new`	`(args) -> new ClassName(args)`

Example:

```java
import java.util.Arrays;
import java.util.List;

public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("alice", "bob", "charlie");

        // Using a lambda to print each name
        names.forEach(s -> System.out.println(s));

        // Using a method reference (more concise and readable)
        // Refers to the println method of the System.out object.
        System.out.println("\nUsing method reference:");
        names.forEach(System.out::println);
    }
}
```

Optional Class

The `java.util.Optional<T>` class is a container object which may or may not contain a non-null value. Its purpose is to provide a better way to handle `null` values, avoiding `NullPointerException`s and making the API clearer about methods that might not return a result.

Stream terminal operations like `reduce()`, `findFirst()`, `max()`, and `min()` return an `Optional` because the stream might be empty.

Key Methods:

- `isPresent()`: Returns `true` if a value is present, `false` otherwise.
- `get()`: Returns the value if present, otherwise throws `NoSuchElementException`.
- `orElse(T other)`: Returns the value if present, otherwise returns a default value.
- `ifPresent(Consumer<T> consumer)`: If a value is present, invokes the specified consumer with the value.

Sort, Distinct, Map, Max, Min, Collect to List

Here is a comprehensive example demonstrating several common stream operations together:

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class CommonStreamOpsExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(10, 4, 7, 2, 8, 4, 7, 10);

        // Chain of operations
        List<Integer> result = numbers.stream()        // 1. Get a stream from the list
            .distinct()                                // 2. Intermediate: Keep only unique elements -> [10, 4, 7, 2, 8]
            .sorted()                                  // 3. Intermediate: Sort them -> [2, 4, 7, 8, 10]
            .map(n -> n * n)                           // 4. Intermediate: Square each number -> [4, 16, 49, 64, 100]
            .collect(Collectors.toList());             // 5. Terminal: Collect the results into a new List

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

        // --- Other Terminal Operations ---
        // Find the maximum value in the original list
        numbers.stream()
            .max(Comparator.naturalOrder())
            .ifPresent(max -> System.out.println("Max value: " + max));

        // Find the minimum value
        numbers.stream()
            .min(Comparator.naturalOrder())
            .ifPresent(min -> System.out.println("Min value: " + min));
    }
}
```

Functional vs. Structural Programming

This is the "what vs. how" distinction in action.

Structural (or Imperative) Programming: You write detailed, step-by-step instructions. You explicitly manage loops, counters, and state.

```
// Structural/Imperative way to find unique even numbers
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
List<Integer> uniqueEvens = new ArrayList<>();
for (Integer number : numbers) {
    if (number % 2 == 0 && !uniqueEvens.contains(number)) {
        uniqueEvens.add(number);
    }
}
System.out.println("Structural result: " + uniqueEvens);
```

Functional (or Declarative) Programming: You declare your intent by composing a pipeline of operations. The underlying implementation is hidden.

```
// Functional/Declarative way using streams
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
List<Integer> functionalResult = numbers.stream()
                                       .filter(n -> n % 2 == 0)
                                       .distinct()
                                       .collect(Collectors.toList());
System.out.println("Functional result: " + functionalResult);
```

The functional approach is often more concise, easier to read (once you're familiar with the syntax), and less prone to bugs related to loop management and mutable state.
