Java Notes

Table of Contents

1.Differences between JVM,JRE,JDK

Imagine you're baking a cake, where:

- **JVM** (Java Virtual Machine) is the oven that runs the baking process.
- **JRE** (Java Runtime Environment) is the kitchen (with ingredients, tools, etc.) needed to bake a cake.
- **JDK** (Java Development Kit) is like the entire kitchen plus the recipe and tools needed for both preparing and baking.

JDK = JRE + Development tools.

1.1 JVM (Java Virtual Machine):

- **What it does:** The JVM is responsible for running the compiled Java code (called **bytecode**). It converts bytecode into machine-readable code that your computer understands.
- **Analogy:** The oven where you bake the cake (runs the code). You don't write or prepare the recipe in the oven, you just use it to cook the cake.

1.2 JRE (Java Runtime Environment):

- **What it does**: JRE contains everything required to run Java applications. It includes the JVM and libraries needed to run Java programs.
- **Analogy**: The kitchen with all the tools (ingredients, bowls, etc.) to prepare the cake, but without the recipe or instructions to create new recipes.

1.3 JDK (Java Development Kit):

- **What it does**: The JDK is a superset of JRE, meaning it includes the JRE plus tools like the **compiler** (javac) and other utilities necessary to develop Java applications.
- **Analogy**: The kitchen (JRE) plus a cookbook and tools to create your own cake recipe. It's what you need to both write and run Java code.

1.4 How it works together:

- **JDK**: You use the JDK to write and compile Java code (writing the recipe).
- **JRE**: Once the code is compiled into bytecode, the JRE provides the environment (ingredients, tools) for running it.
- **JVM**: The JVM runs the compiled bytecode, making sure the computer understands and executes it correctly (oven baking the cake).

2. Variables and Data Types in Java

Variables in Java are containers for storing data values. Every variable in Java must be declared with a specific **data type** that determines the type and size of data it can hold.

2.1 Variables

A **variable** is a named memory location that can store a value. Each variable in Java has:

- **Name**: The identifier by which the variable is referenced (e.g., `age`, `salary`).
- **Type**: Specifies what kind of data the variable can hold (e.g., integers, floating-point numbers, text).
- **Value**: The data stored in the variable.

2.1.1 Types of Variables in Java:

1. **Local Variables**:
    - Declared within a method, constructor, or block.
    - Scope is limited to the block where it is declared.
    - Must be initialized before use.

    Example:

```
public void printNumber() {
    int number = 10; // Local variable
    System.out.println(number);
}
```

2. **Instance Variables**:

- Declared within a class, but outside any method.
- Each instance (object) of the class has its own copy.
- Can be accessed by all methods of the class.
- Example:

```
public class Person {
    String name; // Instance variable
}
```

3. **Static Variables** (Class Variables):

- Declared with the `static` keyword inside a class.
- Shared among all instances of the class.

- Only one copy exists regardless of how many objects are created.
- Example:

```
public class Company {
    static String companyName = "Tech Corp"; // Static variable
}
```

2.2 Data Types

Data types specify the type of data that a variable can hold. In Java, there are two categories:

2.2.1 Primitive Data Types

Java has 8 built-in data types that represent simple values.

Data Type	Size	Default Value	Description	Example
`byte`	1 byte	0	Stores small integer values.	`byte b = 127;`
`short`	2 bytes	0	Stores medium integer values.	`short s = 10000;`
`int`	4 bytes	0	Stores standard integer values.	`int i = 500000;`
`long`	8 bytes	0L	Stores large integer values.	`long l = 100000L;`
`float`	4 bytes	0.0f	Stores decimal numbers with less precision.	`float f = 5.75f;`
`double`	8 bytes	0.0d	Stores decimal numbers with high precision.	`double d = 19.99;`
`boolean`	1 bit	false	Stores true or false values.	`boolean isTrue = true;`
`char`	2 bytes	'\u0000'	Stores a single character.	`char letter = 'A';`

2.2.2 Reference Data Types:

These data types are used to refer to objects. They include:

- **Classes**: Defines objects.
- **Interfaces**: Specifies methods that a class must implement.
- **Arrays**: A collection of variables of the same type.

Example:

```
String name = "Alice";  // Reference type

Person person = new Person();  // Reference to a custom object
```

2.2.3 Primitive vs. Reference Data Types:

- **Primitive Data Types:** Store actual values.
- **Reference Data Types:** Store the address (reference) where the actual object is located in memory.

2.3 Variable Declaration and Initialization

You declare a variable by specifying the data type and name, and optionally initializing it with a value.

Declaration:

```
int age; // Variable declared
```

Initialization:

```
age = 25; // Variable initialized with value 25
```

Declaration + Initialization:

```
int age = 25; // Variable declared and initialized
```

2.4 Type Casting

Type casting is used to convert a variable from one data type to another. There are two types:

- **Implicit Casting (Widening):** Automatic conversion when the data type is widened (e.g., `int` to `double`).
- **Explicit Casting (Narrowing):** Manual conversion when narrowing (e.g., `double` to `int`).

2.4.1 Example of Implicit Casting:

```
int x = 10;

double y = x; // No error, widening from int to double
```

2.4.2 Example of Explicit Casting:

```
double a = 10.5;

int b = (int) a; // Need to cast, narrowing from double to int
```

2.5 Summary

- **Variables** are used to store data, and they must be declared with a specific **data type**.
- Java has **primitive** and **reference** data types.
- Primitive data types include `int`, `double`, `boolean`, etc.
- Reference types refer to objects and arrays.
- You can declare, initialize, and cast variables, converting them between data types when needed.

3. Access Modifiers

3.1 What are Access Modifiers?

- Access modifiers in Java control the **visibility** of classes, methods, and variables. They define **who** can access a particular piece of code. Java has four types of access modifiers: `public`, `private`, `protected`, and **default** (also called package-private).

3.2 Analogy: Office Building Access

Think of an office building with different **rooms** (your methods, variables, or classes). Each room has different levels of **access control** based on **who** should be able to enter.

3.3 Types of Access Modifiers

3.3.1 Public:

- **Analogy: Lobby (Public Area).** Anyone can enter the lobby of a building.
- **Explanation:** If something is `public`, it can be accessed from **anywhere**—inside the same class, other classes, other packages—there are no restrictions.

Example:

```
public class Office {
    public String officeName = "Main Office"; // Can be accessed from anywhere
}
```

3.3.2 Private:

- **Analogy**: **CEO's Office (Private Area)**. Only the CEO (the class itself) can access this room; no one else in the building can.
- **Explanation**: If something is `private`, it can only be accessed **within the same class**. Other classes (even in the same package) **cannot access** it.

Example:

```
public class Office {
    private String secretCode = "1234"; // Only accessible within the Office class
}
```

3.3.3 Protected:

- **Analogy**: **Employee-Only Area**. Only employees and special guests (classes in the same package or subclasses) are allowed in this area.
- **Explanation**: If something is `protected`, it can be accessed by **classes in the same package** and **subclasses** (even if they are in different packages). It's more restrictive than `public` but more open than `private`.

Example:

public class Office {

    protected String employeePassword = "office123"; // Accessible in the same package or through inheritance

}

3.3.4 Default (Package-Private):

- **Analogy**: **Break Room**. Only employees from the same department (package) can access this area, but people from other departments (packages) cannot.
- **Explanation**: If no access modifier is specified, Java treats the element as **package-private**. This means it is accessible by **classes within the same package** but **not by classes in other packages**.

Example:

class Office {

    String officeLocation = "Downtown"; // Accessible only within the same package

}

3.4 Summary of Access Modifiers:

Modifier	Class	Package	Subclass	Global (Other Packages)
Public	✓	✓	✓	✓
Private	✓	✗	✗	✗
Protected	✓	✓	✓	✗ (except subclasses)
Default	✓	✓	✗	✗

3.5 Use Cases:

1. **Public**: Use it when you want your code to be available globally, like a public API or utility method.
2. **Private**: Use it for sensitive data or methods that should not be exposed outside the class.
3. **Protected**: Use it when you want to share access with subclasses or classes within the same package.
4. **Default**: Use it when you want to restrict access to the package level but don't need protection for subclasses.

3.6 Example Combining All Modifiers:

```
public class Building {

    public String buildingName = "Office Tower";    // Accessible from anywhere

    private String securityCode = "9999";           // Only accessible inside this class

    protected String employeeOnlyArea = "3rd Floor"; // Accessible by same package or
subclasses

    String generalRoom = "Lobby";                   // Default (package-private)

}
```

3.7 Conclusion:

- **Access modifiers** are like different **access levels** in a building, determining who can enter certain areas. By controlling access to your code, you protect sensitive data and maintain clean and secure architectures.

4. Static vs. Instance Members in Java

4.1 What are they?

- **Static members** (variables and methods) belong to the class itself rather than any specific instance of the class. This means there is only one copy of the static member, and it is shared across all instances of the class.
- **Instance members** (variables and methods) belong to an individual instance (object) of a class. Each object has its own copy of instance members, which can have different values across different objects.

4.2 Analogy: Library

Think of a **library** where:

- **Static members** represent the library's general rules and resources (like the opening hours, membership fees, and available books). These are the same for every visitor.
- **Instance members** represent the individual **library cards** and personal information of each visitor (like the name, borrowed books, and due dates). These are unique to each visitor.

4.3 Key Points:

1. **Shared vs. Unique**:
   - Static members (like library rules) are shared among all visitors (instances). Everyone follows the same rules.
   - Instance members (like library cards) are unique to each visitor. Each person has their own card with personal details.
2. **Accessing Members**:
   - You access static members using the class name (like looking up the library's rules).
   - You access instance members using an object of the class (like showing your library card to get access to your account).
3. **Memory Allocation**:
   - Static members are allocated memory once when the class is loaded (like having one set of library rules).
   - Instance members are allocated memory each time a new object is created (like each visitor having their own library card).

4.4 Example in Java:

```
class Library {

   // Static member (shared)

   static String libraryName = "City Library";



   // Instance member (unique)

   String visitorName;
```

```
    int visitorID;

    // Constructor
    Library(String name, int id) {
        this.visitorName = name;
        this.visitorID = id;
    }

    // Static method
    static void displayLibraryInfo() {
        System.out.println("Welcome to " + libraryName);
    }

    // Instance method
    void displayVisitorInfo() {
        System.out.println("Visitor Name: " + visitorName + ", Visitor ID: " + visitorID);
    }
}

public class StaticVsInstanceExample {
    public static void main(String[] args) {
        // Accessing static method and member
        Library.displayLibraryInfo();

        // Creating instances (visitors)
        Library visitor1 = new Library("Alice", 101);
        Library visitor2 = new Library("Bob", 102);

        // Accessing instance methods and members
        visitor1.displayVisitorInfo(); // Outputs: Visitor Name: Alice, Visitor ID: 101

        visitor2.displayVisitorInfo(); // Outputs: Visitor Name: Bob, Visitor ID: 102


        // Static member remains the same across all instances

        System.out.println("Library Name (from visitor 1): " + visitor1.libraryName);

        System.out.println("Library Name (from visitor 2): " + visitor2.libraryName);

    }

}
```

4.5 Key Features:

1. **Static Member**: `libraryName` is a static variable shared among all instances of `Library`, representing a common attribute of the library.
2. **Instance Members**: `visitorName` and `visitorID` are instance variables unique to each `Library` object, allowing different visitors to have their own details.
3. **Static Method**: The method `displayLibraryInfo()` can be called without creating an instance of `Library`, demonstrating how static methods can operate on class-level data.

4.6 Conclusion:

- **Static members** are like the general rules and resources of a library that apply to all visitors, while **instance members** are like the personal library cards that contain unique information for each visitor. Understanding this distinction helps in designing classes and managing resources effectively in Java.

5. Constructors

5.1 What is a Constructor?

- A **constructor** in Java is a special method used to **initialize** objects. It is called when an object of a class is created and sets up the initial state of that object.

5.2 Analogy: House Blueprints

Think of a **constructor** as the **blueprint** for building a house. Just like a blueprint defines how to build a house, a constructor defines how to **build and initialize** an object (house). Each time you create a new house (object), you follow the blueprint (constructor) to make sure everything is set up properly, like the number of rooms or the type of materials used.

5.3 Types of Constructors

I. **Default Constructor**:

   - **Analogy: Standard House Blueprint**: If you don't specify anything, the construction team uses a standard blueprint with default specifications (e.g., 2 bedrooms, 1 bathroom).
   - **Explanation**: Java provides a **default constructor** if you don't define one. It initializes the object with default values (like `null` for objects, `0` for integers, etc.).

Example in Java:

```
public class House {
    String type;
    int rooms;

    // Default constructor is provided by Java automatically if you don't define one
}

public class Main {
    public static void main(String[] args) {
        House myHouse = new House(); // Calls the default constructor
        System.out.println(myHouse.type); // Output: null
        System.out.println(myHouse.rooms); // Output: 0
    }
}
```

II. Parameterized Constructor:

- **Analogy**: **Customized House Blueprint**: When building a custom house, you specify things like the number of bedrooms, the size of the kitchen, etc. These parameters help you customize the house to your needs.
- **Explanation**: A **parameterized constructor** allows you to pass arguments to customize the initialization of the object. This is useful when you want to give specific initial values to the object's properties.

Example in Java:

```
public class House {

    String type;

    int rooms;



    // Parameterized constructor to initialize the house with custom values

    public House(String type, int rooms) {

        this.type = type;

        this.rooms = rooms;

    }

}



public class Main {

    public static void main(String[] args) {

        House myHouse = new House("Villa", 4); // Calls the parameterized constructor

        System.out.println(myHouse.type); // Output: Villa
        System.out.println(myHouse.rooms); // Output: 4
    }
}
```

III. Constructor Overloading:

- **Analogy**: **Multiple Blueprints for Different Types of Houses**: A construction company can have different blueprints for various house styles—one for a villa, another for an apartment, etc. You choose the appropriate one based on your needs.
- **Explanation**: You can define **multiple constructors** with different sets of parameters. This is called **constructor overloading**. It allows the creation of objects in different ways, depending on the arguments passed.

Example in Java:

```
public class House {
    String type;
    int rooms;


    // Default constructor
    public House() {
        this.type = "Apartment";
        this.rooms = 2;
    }


    // Parameterized constructor
    public House(String type, int rooms) {
        this.type = type;
        this.rooms = rooms;
    }
}


public class Main {
    public static void main(String[] args) {
        House defaultHouse = new House(); // Calls default constructor
        House customHouse = new House("Villa", 4); // Calls parameterized constructor

        System.out.println(defaultHouse.type); // Output: Apartment
        System.out.println(customHouse.type); // Output: Villa
    }
}
```

5.4 Special Features of Constructors

1. **No Return Type**:
   - A constructor does not have a return type (not even `void`). It just initializes the object and doesn't return any value.
2. **Same Name as the Class**:
   - A constructor always has the same name as the class it belongs to.
3. **Constructor Chaining**:
   - **Analogy**: Sometimes, you might want to use one blueprint to build the foundation, and another one for the specifics. In Java, you can use one constructor inside another constructor using `this()`, which is called **constructor chaining**.

Example:

```
public class House {
    String type;
    int rooms;

    // Constructor chaining
    public House() {
        this("Apartment", 2); // Calls the parameterized constructor
    }

    public House(String type, int rooms) {
        this.type = type;
        this.rooms = rooms;
    }
}
```

5.5 Conclusion:

- A **constructor** is like a **blueprint** for building a house. It sets the **initial state** of an object and ensures that it is properly constructed with either default or specific values. You can have different blueprints (constructors) for various types of houses (objects), making it easy to build and customize your objects based on different needs.

6. Inheritance in Java

6.1 What is it?

- **Inheritance** is a key feature in Java where one class (child class) inherits properties and behaviors (methods) from another class (parent class). It allows code reuse and makes it easier to maintain programs.

6.2 Analogy: Family Traits

Think of a family where children inherit traits from their parents.

6.3 Key Points:

- **Parent Class (Superclass)**: This is like the **parent in a family**. The parent has certain characteristics, like eye color, hair color, or height. In programming, the parent class has variables and methods that can be inherited by its children.
- **Child Class (Subclass)**: This is like the **child in the family**. The child automatically inherits traits from the parent. In Java, the child class inherits methods and properties from the parent class.
- **Code Reuse**: Just like the child doesn't need to "recreate" traits like eye color, the child class doesn't need to rewrite the code that's already in the parent class. The child class can simply use the parent's traits (properties and methods) directly.
- **Extending Behavior**: The child can also have its own unique traits. For example, the child might have a different hobby or skill. Similarly, in Java, the child class can have additional methods or variables that the parent class doesn't have.

6.4 Example:

- **Parent Class (Person)**: A person has common traits like **name**, **age**, and actions like **walking** and **talking**.
- **Child Class (Student)**: A student is a specific type of person. The student **inherits** the name, age, walking, and talking abilities from the parent (Person), but also has some unique traits like **studentID** or **attendClass()** behavior.

```
// Parent class

class Person {

    String name;

    int age;


    void walk() {

        System.out.println(name + " is walking.");

    }
    void talk() {
        System.out.println(name + " is talking.");
    }
}


// Child class (inherits from Person)
class Student extends Person {
    String studentID;


    void attendClass() {
        System.out.println(name + " is attending class.");
    }
}
```

In this example:

- The **Student** class doesn’t need to rewrite the code for walking and talking. It inherits these behaviors from the **Person** class.
- The **Student** class can still add its own specific behavior (like `attendClass()`).

6.5 Important Concepts:

1. **Super and Subclass**: The parent is called the **superclass**, and the child is called the **subclass**.
2. **Method Overriding**: If the child wants to change a trait, like how they talk, they can **override** the method in the parent class.

6.6 Conclusion:

- **Inheritance** makes it easy to create specialized classes by building on existing ones, much like how children inherit traits from their parents but can also have their own unique characteristics.

7. Method Overriding in Java

7.1 What is it?

- **Method overriding** happens when a child class provides its own specific implementation of a method that is already defined in its parent class. It allows a subclass to modify or customize a behavior inherited from its superclass.

7.2 Analogy: Customizing a Family Recipe

Imagine your family has a famous recipe for **pancakes** that your grandmother created. The recipe is passed down to your parents and eventually to you. However, you prefer to make the pancakes a little differently, so you **override** the recipe with your own version while still keeping the base recipe from your family.

7.3 Key Points:

- **Parent's Recipe (Superclass Method):** This is the original pancake recipe that has been used for years. It works well, and everyone in the family uses it by default.
- **Child's Recipe (Overridden Method):** You decide to tweak the recipe. Maybe you add chocolate chips or make the pancakes fluffier. You **override** the original recipe with your own, while still keeping the general idea (pancakes) the same.
- **Same Name, New Behavior:** The overridden method has the **same name** as the one in the parent class but **behaves differently**. In the pancake example, you still call it "making pancakes," but the way you do it is different.

7.4 Example in Java:

- **Parent Class (Animal):** The parent class has a method called `makeSound()` that defines the sound the animal makes.
- **Child Class (Dog):** The child class (Dog) overrides `makeSound()` to customize it with the dog's specific sound (bark), rather than the generic sound from the parent class.

```
// Parent class

class Animal {

    void makeSound() {

        System.out.println("The animal makes a sound.");

    }

}


// Child class (overriding the method)

class Dog extends Animal {

    @Override

    void makeSound() {
        System.out.println("The dog barks.");
    }
}
```

7.5 Key Features:

- **Same Method Signature**: In method overriding, the child class method must have the same **name, return type, and parameters** as the parent class method.
- **@Override Annotation**: In Java, we use the `@Override` annotation to indicate that we are intentionally overriding a method from the parent class.
- **Parent Behavior Access**: If you still want to access the parent's original method (the family's pancake recipe), you can use the `super` keyword to call it:

```
super.makeSound(); // Calls the parent method
```

7.6 Example Output:

- **Parent Method**: If you call `makeSound()` on an **Animal**, it will print `"The animal makes a sound."`
- **Overridden Method**: If you call `makeSound()` on a **Dog**, it will print `"The dog barks."`

7.8 Conclusion:

- **Method overriding** allows child classes to **customize** or modify the behavior of methods they inherit from their parent classes. It's like **taking an inherited family recipe and adjusting it** to fit your taste.

8. Polymorphism in Java

8.1 What is it?

- **Polymorphism** allows objects of different classes to be treated as objects of a common superclass. It means "many forms," and in Java, it allows a single action to behave differently based on the object performing it.

8.2 Analogy: Remote Control for Different Devices

Imagine you have a universal **remote control** that can operate multiple devices, like a TV, air conditioner, or a music system. Although the remote has the same buttons, it works differently based on the device you're controlling.

8.3 Key Points:

1. **Parent Class (Device):** The remote control can be seen as the **common interface** (or parent class) for all devices. It has common buttons like **power, volume**, and **channel.**
2. **Child Classes (TV, AC, Music System):** Each device (TV, AC, music system) is a **child class.** Even though the remote control has the same buttons, the **actions** performed by each button are different for each device:

   - Pressing **Power** turns on the **TV.**
   - Pressing **Power** turns on the **AC.**
   - Pressing **Power** turns on the **Music System.**

3. **Polymorphism in Action:** Even though you are using the same remote (method), it **behaves differently** depending on the device (object) you are controlling.

8.4 Example in Java:

1. **Parent Class (Animal):** You have a method `makeSound()` that is common to all animals.
2. **Child Classes (Dog, Cat, Cow):** Each animal makes a different sound. Polymorphism allows you to call `makeSound()` on any animal object, and it will respond with its own sound.

```
// Parent class

class Animal {

    void makeSound() {

        System.out.println("The animal makes a sound.");

    }

}


// Child class - Dog

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

// Child class - Cat
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("The cat meows.");
    }
}

// Child class - Cow
class Cow extends Animal {
    @Override
    void makeSound() {
        System.out.println("The cow moos.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Parent class object
        Animal myDog = new Dog();        // Child class object - Dog
        Animal myCat = new Cat();        // Child class object - Cat
        Animal myCow = new Cow();        // Child class object - Cow
        myAnimal.makeSound();  // Outputs: The animal makes a sound.
        myDog.makeSound();     // Outputs: The dog barks.
        myCat.makeSound();     // Outputs: The cat meows.
        myCow.makeSound();     // Outputs: The cow moos.
    }
}
```

8.5 Key Features:

- **Method Overriding**: Polymorphism is achieved through **method overriding**. Each child class overrides the `makeSound()` method to give it its own unique behavior.
- **Single Interface, Multiple Behaviors**: The `makeSound()` method behaves differently based on the object (dog, cat, or cow), even though you call the same method.
- **Flexible Code**: Polymorphism allows you to write flexible code. You don't need to know which specific animal you're working with, just that it's an **Animal**. This makes your program easier to extend or modify later.

8.6 Conclusion:

- **Polymorphism** allows one interface (like a remote control) to handle multiple forms of objects (different devices). In Java, it means that methods can take many forms, making the program more dynamic and flexible. It's like using one remote for various devices, but each responds differently depending on the type of device.

9. Encapsulation in Java

9.1 What is it?

- **Encapsulation** is one of the four fundamental Object-Oriented Programming (OOP) principles. It is the practice of wrapping data (variables) and methods (functions) that operate on the data into a single unit or class, and restricting access to some of the object's components. This helps protect the object's integrity and maintain its state.

9.2 Analogy: Capsule or Container

Think of **encapsulation** like a **capsule** or a **container.** Just like a capsule holds different ingredients inside, encapsulation groups data and methods together. You can only access the contents of the capsule in certain ways (like taking medicine), ensuring that the ingredients remain safe and functional.

9.3 Key Points:

1. **Data Hiding**: The contents of the capsule (data) are hidden from the outside world. You cannot directly access the ingredients; you have to use the provided methods (like opening the capsule) to interact with them.
2. **Public and Private Access**:
    - **Private Data**: Inside the capsule, ingredients are marked as private. This means they can't be accessed directly from outside the capsule.
    - **Public Methods**: You provide public methods (like a lid or opening) that allow controlled access to the data.
3. **Maintaining Integrity**: Encapsulation ensures that the data remains valid. You can add checks or conditions within the public methods to validate the data before using it.

9.4 Example in Java:

```
// Class definition

class BankAccount {

   // Private data members

   private String accountNumber;

   private double balance;


   // Constructor

   public BankAccount(String accountNumber, double initialBalance) {

      this.accountNumber = accountNumber;

      this.balance = initialBalance;

   }
```

```
    // Public method to deposit money
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount; // Add amount to balance
            System.out.println("Deposited: " + amount);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }

    // Public method to withdraw money
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount; // Deduct amount from balance
            System.out.println("Withdrew: " + amount);
        } else {
            System.out.println("Invalid withdrawal amount.");
        }
    }

    // Public method to check balance
    public double getBalance() {
        return balance; // Return current balance
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        // Creating a BankAccount object
        BankAccount myAccount = new BankAccount("123456789", 500.00);

        // Accessing methods to manipulate the account
        myAccount.deposit(200);
        myAccount.withdraw(100);

        // Accessing balance through a public method
        System.out.println("Current Balance: " + myAccount.getBalance());

        // Trying to access private data directly (this will cause an error)
        // System.out.println(myAccount.balance); // Error: balance has private access
    }
}
```

9.5 Key Features:

1. **Private Data Members:** `accountNumber` and `balance` are private, so they cannot be accessed directly outside the class.
2. **Public Methods:** Methods like `deposit()`, `withdraw()`, and `getBalance()` are public, providing controlled access to the private data.
3. **Data Validation:** The `deposit()` and `withdraw()` methods include validation checks, ensuring that only valid amounts can be deposited or withdrawn.

9.6 Conclusion:

- **Encapsulation** keeps the data safe from outside interference and misuse. By using encapsulation, you ensure that your object can maintain its internal state and provide a clear interface for interaction. It's like having a capsule that securely holds its contents, allowing only controlled access to ensure everything works as intended.

10. Abstraction in Java

10.1 What is it?

- **Abstraction** is one of the four fundamental principles of Object-Oriented Programming (OOP). It involves hiding the complex implementation details and showing only the essential features of an object. Abstraction helps to reduce programming complexity and increase efficiency.

10.2 Analogy: TV Remote Control

Think of **abstraction** like using a **TV remote control**. When you want to watch TV, you only need to know how to use the remote buttons (like power, volume, and channel). You don't need to understand how the TV works internally or how the signals are processed. The remote provides a simplified interface to control the TV.

10.3 Key Points:

1. **Hiding Complexity**: Abstraction allows you to focus on what an object does rather than how it does it. Just like with a remote, you focus on the functions (like changing the channel) without needing to know the internal workings of the TV.
2. **Essential Features**: In abstraction, only the essential features are visible to the user. The complex details are hidden, allowing users to interact with the object in a straightforward way.
3. **Interfaces and Abstract Classes**: In Java, abstraction is achieved using interfaces and abstract classes, which define methods that must be implemented by subclasses.

10.4 Example in Java:

```
// Abstract class

abstract class Shape {

   // Abstract method (no implementation)

   abstract void draw();

}


// Subclass (inherits from Shape)

class Circle extends Shape {

   // Providing implementation for the abstract method

   void draw() {

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

   }

}
```

```
// Subclass (inherits from Shape)
class Rectangle extends Shape {
    // Providing implementation for the abstract method
    void draw() {
        System.out.println("Drawing a rectangle.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Shape myCircle = new Circle();     // Creating a Circle object
        Shape myRectangle = new Rectangle(); // Creating a Rectangle object

        myCircle.draw();       // Outputs: Drawing a circle.
        myRectangle.draw();    // Outputs: Drawing a rectangle.
    }
}
```

10.5 Key Features:

- **Abstract Class:** `Shape` is an abstract class that contains an abstract method `draw()`. It does not provide an implementation for this method, leaving it to the subclasses.
- **Subclass Implementation:** The subclasses (`Circle`, `Rectangle`) provide their own implementations of the `draw()` method. This is similar to how different TV models may have different features while all providing the basic function of turning on/off.
- **Simplified Interaction:** The user interacts with the `Shape` type without needing to know the details of how each shape is drawn. Just as you use the remote without knowing how the TV processes the commands.

10.6 Conclusion:

- **Abstraction** allows you to hide the complex implementation details of an object and expose only the necessary functionalities, making it easier to interact with objects. It helps to reduce complexity and enhances the usability of your code, much like a TV remote simplifies controlling the TV.

11. Interfaces in Java

11.1 What is it?

- An **interface** in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors. They are used to achieve abstraction and multiple inheritance in Java, allowing classes to implement the same set of methods.

11.2 Analogy: Contract

Think of an **interface** as a **contract**. When you sign a contract, you agree to perform certain tasks or uphold specific conditions, but the contract itself doesn't specify how you will accomplish those tasks. Different people or companies can fulfill the contract in their own way while still adhering to the agreed-upon terms.

11.3 Key Points:

- **Defining a Contract**: An interface defines a contract that classes can implement. The interface specifies what methods must be included, but it does not provide any implementation details.
- **Multiple Implementations**: Just as different parties can fulfill the same contract in various ways, multiple classes can implement the same interface with their unique behaviors.
- **Abstraction**: Interfaces help achieve abstraction by separating the definition of methods from their implementation.

11.4 Example in Java:

```
// Interface definition

interface Animal {

    // Abstract method

    void makeSound(); // No implementation provided

}


// Class implementing the interface

class Dog implements Animal {

    // Providing implementation for the interface method

    public void makeSound() {

        System.out.println("The dog barks.");

    }

}
```

```
// Class implementing the interface
class Cat implements Animal {
    // Providing implementation for the interface method
    public void makeSound() {
        System.out.println("The cat meows.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Creating a Dog object
        Animal myCat = new Cat(); // Creating a Cat object

        myDog.makeSound(); // Outputs: The dog barks.
        myCat.makeSound(); // Outputs: The cat meows.
    }
}
```

11.5 Key Features:

- **Interface Definition**: The `Animal` interface defines the `makeSound()` method without any implementation, acting like a contract that requires any implementing class to define this method.
- **Implementation**: The `Dog` and `Cat` classes implement the `Animal` interface and provide their unique implementations of the `makeSound()` method.
- **Multiple Implementations**: You can create different classes (like `Dog`, `Cat`, or any other animal) that implement the `Animal` interface, each providing its behavior while adhering to the same contract.

11.6 Conclusion:

- **Interfaces** allow you to define a contract that multiple classes can implement, ensuring they provide specific functionalities. This promotes flexibility and reusability in your code, as different classes can fulfill the same contract in various ways, just like different parties can fulfill the terms of a contract according to their unique methods.

12. Differences Between Interface and Abstract Class

12.1 Purpose

- **Interface:** An interface defines a contract for what a class can do, without specifying how it does it. It is primarily used to achieve abstraction and multiple inheritance.
- **Abstract Class:** An abstract class can have both abstract methods (without implementation) and concrete methods (with implementation). It serves as a base class that can provide some default behavior while still enforcing that certain methods be implemented by subclasses.

12.2 Method Definitions

- **Interface:** All methods in an interface are implicitly abstract (unless they are default or static methods), meaning they do not have a body.
- **Abstract Class:** An abstract class can have abstract methods (without implementation) as well as fully defined methods (with implementation).

12.3 Multiple Inheritance

- **Interface:** A class can implement multiple interfaces, allowing for a form of multiple inheritance.
- **Abstract Class:** A class can only extend one abstract class, adhering to single inheritance.

12.4 Fields

- **Interface:** Cannot have instance variables (fields). However, it can have static final variables (constants).
- **Abstract Class:** Can have instance variables, providing a way to store state.

12.5 Constructor

- **Interface:** Cannot have constructors, as they cannot be instantiated.
- **Abstract Class:** Can have constructors, allowing for initialization of state when a subclass is instantiated.

12.6 Accessibility Modifiers

- **Interface:** All methods in an interface are public by default, and you cannot use access modifiers like private or protected for methods.
- **Abstract Class:** Can have methods with different access modifiers (public, protected, or private).

12.7 Implementation

- **Interface:** A class that implements an interface must provide implementations for all of its methods.

- **Abstract Class:** A class that extends an abstract class may implement only the abstract methods, but it is not required to implement all the methods, as it can still inherit concrete methods from the abstract class.

12.8 Summary Table

Feature	Interface	Abstract Class
Purpose	Defines a contract	Provides a base class with some behavior
Method Definitions	All methods are abstract	Can have both abstract and concrete methods
Multiple Inheritance	Supports multiple inheritance	Supports single inheritance
Fields	No instance variables (only constants)	Can have instance variables
Constructor	No constructors	Can have constructors
Accessibility Modifiers	All methods are public	Can use different access modifiers
Implementation	Must implement all methods	May implement only abstract methods

12.9 Example to Illustrate the Differences

```
// Interface

interface Animal {

    void makeSound(); // No implementation

}


// Abstract Class

abstract class Mammal {

    // Abstract method

    abstract void walk();


    // Concrete method

    void breathe() {
        System.out.println("Breathing air.");
    }
}

// Class implementing the interface
class Dog extends Mammal implements Animal {
    public void makeSound() {
        System.out.println("The dog barks.");
    }

    public void walk() {
        System.out.println("The dog walks on four legs.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound(); // Outputs: The dog barks.
        myDog.walk();      // Outputs: The dog walks on four legs.
        myDog.breathe();   // Outputs: Breathing air.
    }
}
```

12.10 Conclusion

- **Interfaces** are used when you want to define a contract that can be implemented by multiple classes, allowing for multiple inheritance.
- **Abstract classes** are used when you want to provide a common base with shared behavior and enforce specific methods that subclasses must implement.

13. Method Overloading

13.1 What is Method Overloading?

- **Method overloading** allows a class to have **multiple methods with the same name** but **different parameter lists** (type, number, or order of parameters). This enables you to perform **similar actions** in different ways.

13.2 Analogy: Swiss Army Knife

Think of method overloading as a **Swiss Army Knife.** Even though the knife looks like a single tool, it has **multiple tools** inside it that can handle different tasks (like a blade, scissors, screwdriver, etc.). Each tool is designed for a specific job, but they all belong to the same Swiss Army Knife.

Similarly, in method overloading, you have **one method name** but multiple **versions** of it that can handle different parameter combinations.

13.3 Key Features of Method Overloading:

- **Same Method Name, Different Parameters:**

All the overloaded methods share the **same name**, but each has a **different parameter list.** These differences can be in the **number of parameters, types of parameters**, or the **order** of parameters.

- **Return Type Doesn't Matter:**

Method overloading **does not depend on the return type** of the method. You cannot overload methods just by changing their return type.

13.4 Examples:

I. Overloading Based on Number of Parameters:

- **Analogy:** Imagine using a knife to cut both small and large objects. The same tool is used, but with different effort for each size.
- **Explanation:** Here, the methods have the same name but take different numbers of parameters.

Example in Java:

```
public class Calculator {
    // Method to add two numbers
    public int add(int a, int b) {
        return a + b;
    }
    // Overloaded method to add three numbers

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

        return a + b + c;

    }

}


public class Main {

    public static void main(String[] args) {

        Calculator calc = new Calculator();

        System.out.println(calc.add(10, 20)); // Output: 30

        System.out.println(calc.add(10, 20, 30)); // Output: 60

    }

}
```

II. Overloading Based on Parameter Types:

- **Analogy**: The Swiss Army Knife can handle different tasks (cutting paper, cutting wood), but the **tool used depends on the material.** You use scissors for paper and a blade for wood.
- **Explanation**: Methods have the same name but accept different types of parameters.

Example in Java:

```
public class Calculator {

    // Method to add two integers

    public int add(int a, int b) {

        return a + b;

    }


    // Overloaded method to add two doubles

    public double add(double a, double b) {

        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(10, 20)); // Output: 30
        System.out.println(calc.add(10.5, 20.5)); // Output: 31.0
    }
}
```

III. Overloading Based on Parameter Order:

- **Analogy**: Using the Swiss Army Knife scissors to cut either left-handed or right-handed. The action is the same, but the order in which you approach the task changes.
- **Explanation**: Methods have the same name and types of parameters, but the order of the parameters is different.

Example in Java:

```
public class Calculator {
    // Method to print integer first, then double
    public void print(int a, double b) {
        System.out.println("Integer: " + a + ", Double: " + b);
    }

    // Overloaded method to print double first, then integer
    public void print(double a, int b) {
        System.out.println("Double: " + a + ", Integer: " + b);
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.print(10, 20.5); // Output: Integer: 10, Double: 20.5
        calc.print(20.5, 10); // Output: Double: 20.5, Integer: 10
    }
}
```

13.5 Benefits of Method Overloading:

1. **Cleaner Code**:
   - Method overloading allows you to use the **same method name** for **similar functionality,** making the code cleaner and more readable.
2. **Improves Readability**:
   - Having multiple methods with the same name reduces the need to remember multiple method names, enhancing readability and ease of use.
3. **Flexibility**:
   - Method overloading provides flexibility to call the same method with different parameter types or numbers without having to create completely new methods.

13.6 Summary of Method Overloading:

- **Same name, different parameters**: You can have multiple methods with the same name but different parameter lists.
- **Return type doesn't matter**: Method overloading is determined by the parameter list, not the return type.
- **Cleaner and more readable**: It improves the structure and readability of your code by allowing similar operations to share a common method name.

13.7 Quick Overview:

- **Analogy**: Method overloading is like a **Swiss Army Knife**—one tool, many functionalities.
- **Overloading Criteria**: Differ in the **number, type,** or **order** of parameters.

14. Wrapper Classes

14.1 What are Wrapper Classes?

- **Wrapper classes** are used to "wrap" or **encapsulate primitive data types** (like `int`, `char`, etc.) into **objects**. Java provides wrapper classes for all primitive types. These classes enable you to treat primitives as objects.

14.2 Analogy: Wrapper as a Gift Box

Imagine you have a small object, like a **toy**, and you want to **wrap** it inside a **gift box** to make it presentable or easier to handle.

- The **toy** represents the **primitive data type** (simple and direct, like `int` or `char`).
- The **gift box** is the **wrapper class** that wraps around the toy, turning it into an object, making it more flexible for certain situations (like storing it in collections or passing it to methods that require objects).

14.3 Why Use Wrapper Classes?

1. **Collections:** Java **collections** (like `ArrayList`, `HashMap`) cannot store primitive types directly. Instead, they store objects. Wrapper classes allow you to store primitive values in these collections by turning them into objects.
2. **Object Methods:** Primitive types don't have methods. Wrapping them in objects allows you to use methods like `.toString()`, `.compareTo()`, etc.

14.4 Primitive Types and Their Wrapper Classes:

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

14.4.1 Example:

Using a **primitive `int`** vs. an **`Integer` wrapper class**:

Without Wrapper Class:

int number = 10; // This is a primitive type

System.out.println(number); // It's a simple integer value, no object methods

With Wrapper Class:

Integer number = 10; // Wrapped the primitive int in an Integer object

System.out.println(number.toString()); // Now we can use object methods like toString()

14.5 Features of Wrapper Classes:

1. **Boxing (Wrapping)**: Converting a primitive value into an object of its corresponding wrapper class.
    - **Analogy:** Taking a toy and putting it inside a gift box.

Example:

int num = 10;

Integer wrappedNum = Integer.valueOf(num); // Boxing (or wrapping) the int into an Integer object

2. **Unboxing (Unwrapping)**: Converting a wrapper class object back into its corresponding primitive type.

- **Analogy**: Taking the toy out of the gift box.

Example:

Integer wrappedNum = 10;

int num = wrappedNum.intValue(); // Unboxing (or unwrapping) the Integer object back into a primitive int

3. **Autoboxing and Unboxing**: Java automatically handles boxing and unboxing. You don't need to manually convert between primitives and wrapper objects.

- **Autoboxing**: Automatically converting a primitive into its corresponding wrapper object.

int num = 5;

```
Integer wrappedNum = num;  // Java automatically boxes it into an Integer object
```

- **Unboxing:** Automatically converting a wrapper object into its corresponding primitive.

```
Integer wrappedNum = 10;

int num = wrappedNum;  // Java automatically unboxes it into a primitive int
```

14.6 Example in Practice:

```
import java.util.ArrayList;


public class Main {

    public static void main(String[] args) {

        // Create an ArrayList to store Integer objects

        ArrayList<Integer> numbers = new ArrayList<>();


        // Add primitive int values (Autoboxing happens here)

        numbers.add(10);  // Autoboxes int 10 into Integer(10)

        numbers.add(20);  // Autoboxes int 20 into Integer(20)


        // Retrieve elements (Unboxing happens here)

        int firstNum = numbers.get(0);  // Unboxes Integer(10) into int 10


        // Output the result

        System.out.println(firstNum);  // Output: 10

    }

}
```

14.7 Benefits of Wrapper Classes:

1. **Collections Compatibility:** You can store primitive values in Java collections like `ArrayList`, `HashMap`, etc.
2. **Method Support:** Wrapper classes come with useful methods (`toString()`, `compareTo()`, etc.) that you can't use with primitive types.

3. **Flexibility in Coding**: Wrappers provide more flexibility when passing values into methods that require objects.

14.8 Summary of Wrapper Classes:

- **Analogy**: Primitive types are like toys; wrapper classes are like gift boxes for those toys.
- **Boxing**: Wrapping a primitive into its corresponding object (toy into gift box).
- **Unboxing**: Unwrapping the object back into a primitive (taking the toy out of the box).
- **Autoboxing/Unboxing**: Java's automatic handling of boxing and unboxing.

15. Error Handling in Java

15.1 What is it?

- **Error handling** in programming refers to the process of responding to and managing errors that occur during the execution of a program. It allows developers to gracefully handle unexpected situations, ensuring that the program can continue running or terminate safely without crashing.

15.2 Analogy: Lifeguard at a Pool

Think of **error handling** like a **lifeguard at a pool.** The lifeguard's job is to watch for any problems (like someone struggling in the water) and respond appropriately to ensure everyone's safety.

15.3 Key Points:

- **Monitoring for Issues:** Just as a lifeguard keeps an eye on swimmers for signs of trouble, a program monitors for exceptions or errors during execution.
- **Prepared Response:** When a lifeguard sees someone in distress, they have a plan to help them (like jumping in or throwing a flotation device). Similarly, when a program encounters an error, it uses error handling constructs (like try-catch blocks) to respond.
- **Preventing Crashes:** A lifeguard helps prevent serious accidents by intervening when needed. In programming, effective error handling prevents the entire program from crashing due to unexpected errors.

15.4 Example in Java:

```
public class Lifeguard {

    public void monitorSwimmer(String swimmer) {

        try {

            // Simulate checking if the swimmer is safe

            if (swimmer.equals("in trouble")) {

                throw new Exception("Swimmer is struggling!");

            }

            System.out.println(swimmer + " is swimming safely.");

        } catch (Exception e) {

            // Lifeguard's response to the error

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

            rescue(swimmer); // Rescues the swimmer in trouble

        }

    }
    private void rescue(String swimmer) {
        System.out.println("Lifeguard is rescuing " + swimmer + "!");
    }

    public static void main(String[] args) {
        Lifeguard lifeguard = new Lifeguard();
        lifeguard.monitorSwimmer("in trouble"); // Simulates a swimmer in distress
        lifeguard.monitorSwimmer("swimming safely"); // Simulates a swimmer safe
    }
}
```

15.5 Key Features:

- **Try-Catch Block**: In the example, the `try` block checks for an error (the swimmer in trouble). If an error occurs, the `catch` block handles it.
- **Graceful Handling**: When an error is detected, the lifeguard (program) responds by rescuing the swimmer instead of letting the situation escalate.
- **Maintaining Control**: Just as a lifeguard helps maintain a safe environment, error handling keeps the program running smoothly despite unexpected issues.

15.6 Conclusion:

- **Error handling** is like having a lifeguard at a pool, ensuring that any problems are monitored and handled effectively. It allows programs to respond gracefully to unexpected situations, preventing crashes and ensuring user safety.

16 Exception Handling in Java

16.1 What is it?

- **Exception Handling** is a mechanism in Java that allows developers to manage errors and exceptional conditions in a controlled manner. It helps prevent the program from crashing and provides a way to respond to errors gracefully.

16.2 Analogy: Safety Net

Think of **exception handling** like having a **safety net** in a circus. When acrobats perform risky stunts high above the ground, there's a safety net below. If they fall, the net catches them, preventing serious injury. Similarly, exception handling helps catch and manage errors in your code, ensuring that your program can recover gracefully instead of crashing.

16.3 Key Points:

- **Catching Errors**: Just as the safety net catches falling acrobats, exception handling catches errors (exceptions) that occur during the execution of a program.
- **Graceful Recovery**: If an acrobat falls, the safety net allows them to get back up and continue the show. In programming, exception handling allows the program to handle the error and continue executing without crashing.
- **Prevention of Crashes**: The safety net prevents acrobats from hitting the ground hard, which could cause injuries. Exception handling prevents the program from terminating unexpectedly due to unhandled errors.

16.4 Example in Java:

```
public class ExceptionHandlingExample {

    public static void main(String[] args) {

        try {

            // Code that may throw an exception

            int result = divide(10, 0);

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

        } catch (ArithmeticException e) {

            // Handling the exception

            System.out.println("Error: Cannot divide by zero!");

        } finally {

            // This block will execute regardless of whether an exception occurred or not

            System.out.println("Cleanup can be done here.");

        }

    }
    // Method that can throw an exception
    static int divide(int a, int b) {
        return a / b; // This will throw ArithmeticException if b is 0
    }
}
```

16.5 Key Features:

- **Try Block:** The `try` block contains code that may throw an exception (like performing a risky stunt). If an exception occurs, control is transferred to the `catch` block.
- **Catch Block:** The `catch` block handles the exception (like the safety net). In this example, if a division by zero occurs, it catches the `ArithmeticException` and prints an error message.
- **Finally Block:** The `finally` block executes regardless of whether an exception was thrown or handled. It's a place to perform cleanup tasks, similar to how a safety net is always present, regardless of whether an acrobat falls or not.

16.6 Conclusion:

Exception handling in Java is like a **safety net** in a circus that catches errors and allows the program to recover gracefully. It helps manage unexpected situations, ensuring that the program runs smoothly without crashing.

17. Garbage Collection in Java

17.1 What is it?

- **Garbage Collection (GC)** is Java's way of automatically finding and cleaning up unused objects in memory, so the system doesn't run out of space. Java automatically handles memory management, so you don't need to worry about deleting unused objects yourself.

17.2 Analogy: Room Cleanup After a Party

- Imagine you've just had a big party in your living room, and now there are cups, plates, and wrappers scattered everywhere. After the party, the **Garbage Collector** is like a **cleaner** who comes in and removes the items that are no longer in use (empty cups, dirty plates).

17.3 Key points:

1. **Objects are like cups and plates** left after a party. You use them while needed.
2. **Garbage Collector is like the cleaner.** They check the room (your program) for stuff that is no longer being used and clean it up (remove unused objects).
3. **Automatic cleanup:** You don't have to ask the cleaner to remove each cup; they automatically know when something is no longer needed.
4. **Memory management:** If the cleaner didn't show up, the room would get too cluttered with garbage. Similarly, without garbage collection, your program would eventually run out of memory.

17.4 Stages of Garbage Collection:

- **Marking:** The cleaner checks which items are still being used (like cups still being used by guests) and which are not (empty plates, trash).
- **Sweeping:** The cleaner removes all the unused items.

So, **Java's Garbage Collection** is like a cleaner who ensures your room (program's memory) doesn't get cluttered, making sure everything runs smoothly!

18. Multithreading in Java

18.1 What is it?

- **Multithreading** allows a program to perform multiple tasks (threads) at the same time. Each thread runs independently but shares the same resources, like memory. This improves the performance of a program by allowing it to do several things at once.

18.2 Analogy: Making a Sandwich with Multiple Tasks

Imagine you're making a sandwich, and you want to get it done faster by doing **multiple tasks at the same time** instead of doing them one after another.

18.3 Key Points:

1. **Thread:** Each task, like spreading butter, cutting veggies, or toasting bread, is a **thread.** Each task runs in parallel (at the same time).
2. **Main Task:** The overall goal (making the sandwich) is like the **main thread.** It controls the process, but the smaller tasks help finish it faster.
3. **Multithreading:** Instead of doing each task **one after another** (first buttering, then cutting veggies, etc.), you can **multitask** (butter the bread while the toaster is working). These are **multiple threads** running simultaneously.
4. **Sharing Resources:** All the tasks (threads) share the same kitchen and ingredients (memory/resources), so they must be managed carefully to avoid conflict (e.g., two tasks trying to use the knife at the same time).

18.4 Example:

- **Single-threaded program**: You first butter the bread, then toast it, and finally cut the veggies. You do each task one by one. It’s slow because you’re not using all your time efficiently.
- **Multithreaded program**: While the bread is toasting, you can butter the second slice and start cutting veggies. Multiple tasks run simultaneously, making the sandwich faster!

18.5 Important Concepts:

1. **Synchronization:** If two people are making different parts of the sandwich, they must coordinate who uses the knife (shared resource). This is called **synchronization** in multithreading, ensuring that threads don’t interfere with each other.
2. **Concurrency:** All tasks (threads) seem to be happening at the same time (like buttering, toasting, cutting veggies). This is called **concurrency.**

18.6 Conclusion:

- Multithreading helps Java programs run faster by doing **several tasks at once** rather than waiting for one task to finish before starting another. Just like you save time by buttering bread while the toaster is working, multithreading improves efficiency by managing multiple tasks in parallel.

19. Concurrency in Java

19.1 What is it?

- **Concurrency** is the ability of a program to execute multiple tasks or threads simultaneously, making efficient use of resources and improving performance. In Java, this is achieved through multi-threading, where different threads can run independently but may interact with shared resources.

19.2 Analogy: Busy Restaurant

Think of **concurrency** like a **busy restaurant.** In a restaurant, multiple customers are being served at the same time by several waiters. Each waiter can take orders, serve food, and handle payments independently, ensuring that all customers are attended to promptly.

19.3 Key Points:

- **Multiple Tasks**: Just as multiple customers can be served at once, a program can execute several threads simultaneously to perform different tasks.
- **Independent Workers**: Each waiter (thread) can work independently, taking care of their assigned customers (tasks) without interfering with one another.
- **Shared Resources**: Waiters may need to share resources, like the kitchen (data), to prepare food. In programming, threads may share resources, leading to potential issues like race conditions if not managed properly.

19.4 Example in Java:

```
class Restaurant {

    public void serveCustomer(String customer) {

        System.out.println("Serving " + customer);

    }

}


class Waiter extends Thread {

    private String customer;

    private Restaurant restaurant;


    public Waiter(Restaurant restaurant, String customer) {

        this.restaurant = restaurant;

        this.customer = customer;

    }
    public void run() {
        restaurant.serveCustomer(customer);
    }
}

public class BusyRestaurant {
    public static void main(String[] args) {
        Restaurant restaurant = new Restaurant();

        // Creating multiple waiters (threads)
        Waiter waiter1 = new Waiter(restaurant, "Customer 1");
        Waiter waiter2 = new Waiter(restaurant, "Customer 2");
        Waiter waiter3 = new Waiter(restaurant, "Customer 3");

        // Starting the threads (waiters)
        waiter1.start();
        waiter2.start();
        waiter3.start();

        // Wait for threads to finish
        try {
            waiter1.join();
            waiter2.join();
            waiter3.join();
        } catch (InterruptedException e) {
            System.out.println("A waiter was interrupted!");
        }

        System.out.println("All customers have been served!");
    }
}
```

19.5 Key Features:

- **Multiple Threads:** In the example, each `Waiter` represents a thread serving a different customer concurrently.
- **Independent Execution:** Each waiter can start serving a customer at the same time, demonstrating concurrent execution.
- **Resource Sharing:** All waiters share the same `Restaurant` object, simulating how threads share resources.

19.6 Conclusion:

- **Concurrency** is like a busy restaurant where multiple waiters serve customers simultaneously. It enhances the efficiency of a program by allowing multiple tasks to run independently while managing shared resources effectively.

20. Java Collections Framework

20.1 What is it?

- The **Java Collections Framework** is a set of classes and interfaces that provide various data structures and algorithms to store, manipulate, and retrieve data efficiently. It allows developers to work with groups of objects, making data management easier.

20.2 Analogy: Toolbox

Think of the **Java Collections Framework** as a **toolbox**. Just as a toolbox contains different types of tools (like hammers, screwdrivers, and wrenches) for various tasks, the Java Collections Framework provides different data structures (like lists, sets, and maps) for managing collections of objects.

20.3 Key Points:

1. **Different Tools for Different Tasks:**
    - A toolbox has specific tools for specific jobs (e.g., a hammer for nails, a screwdriver for screws). Similarly, the Collections Framework provides different types of collections suited for different purposes.
2. **Organized and Efficient:**
    - A toolbox keeps your tools organized and makes it easy to find the right tool when you need it. The Collections Framework organizes data and provides efficient methods to retrieve and manipulate that data.
3. **Interchangeable Tools:**
    - Just as you can replace one tool with another depending on the task (e.g., using a wrench instead of a screwdriver), you can choose different types of collections based on your requirements (e.g., using an `ArrayList` for dynamic arrays or a `HashSet` for unique elements).

20.4 Types of Collections:

1. **List**:
    - **Analogy**: Like a drawer with tools arranged in a specific order.
    - **Example**: `ArrayList`, `LinkedList`
    - **Usage**: Used for storing ordered collections that allow duplicates.
2. **Set**:
    - **Analogy**: Like a container that holds only unique tools, without duplicates.
    - **Example**: `HashSet`, `TreeSet`
    - **Usage**: Used for storing unique elements with no duplicates.
3. **Map**:
    - **Analogy**: Like a labeled organizer where each tool has a specific label (key) associated with it.
    - **Example**: `HashMap`, `TreeMap`
    - **Usage**: Used for storing key-value pairs, where each key maps to a specific value.

20.5 Example in Java:

```
import java.util.ArrayList;

import java.util.HashSet;

import java.util.HashMap;


public class CollectionsExample {

    public static void main(String[] args) {

        // List Example

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

        toolsList.add("Hammer");

        toolsList.add("Screwdriver");

        toolsList.add("Wrench");

        System.out.println("Tools List: " + toolsList);


        // Set Example

        HashSet<String> uniqueTools = new HashSet<>();

        uniqueTools.add("Hammer");

        uniqueTools.add("Screwdriver");

        uniqueTools.add("Hammer"); // Duplicate won't be added

        System.out.println("Unique Tools: " + uniqueTools);


        // Map Example

        HashMap<String, String> toolBox = new HashMap<>();

        toolBox.put("Hammer", "Used for driving nails");

        toolBox.put("Screwdriver", "Used for turning screws");

        System.out.println("Tool Descriptions: " + toolBox);

    }

}
```

20.6 Key Features:

1. **ArrayList:** A list that maintains the order of insertion and allows duplicates (like a drawer where tools can be arranged in order).
2. **HashSet:** A set that stores unique elements and ignores duplicates (like a container that only holds unique tools).
3. **HashMap:** A map that associates keys with values (like an organizer with labeled sections for each tool).

20.7 Differences between their implementations

1. List vs. Set vs. Map

Feature	**List**	**Set**	**Map**
Definition	Ordered collection of elements	Unordered collection of unique elements	Collection of key-value pairs
Duplicates	Allows duplicates	No duplicates	Keys must be unique, but values can repeat
Order	Maintains insertion order	Doesn't guarantee order	Doesn't guarantee order (except `LinkedHashMap`)
Access	Can access by index (e.g., `list.get(0)`)	Access by element (no index)	Access via keys (`map.get(key)`)
Implementations	`ArrayList`, `LinkedList`, `Vector`	`HashSet`, `LinkedHashSet`, `TreeSet`	`HashMap`, `TreeMap`, `LinkedHashMap`

2. List Implementations

Feature	ArrayList	LinkedList	Vector
Structure	Resizable array	Doubly-linked list	Synchronized resizable array
Performance	Fast random access, slower insertions/removals (O(n) for shifting elements)	Fast insertions/removals at start/middle but slow random access (O(n))	Similar to `ArrayList`, but thread-safe
Use Cases	Best for frequent random access	Best for frequent insertions/removals	Thread-safe, legacy class (use with care)

3. Set Implementations

Feature	**HashSet**	**LinkedHashSet**	**TreeSet**
Ordering	No ordering	Maintains insertion order	Sorted in natural order or by comparator
Performance	O(1) for add, remove, and contains	O(1) with insertion order	O(log n) for add, remove, and contains
Use Cases	When you need no duplicates, no ordering	When insertion order matters	When sorted data is required

4. Map Implementations

Feature	**HashMap**	**LinkedHashMap**	**TreeMap**
Ordering	No ordering	Maintains insertion order	Sorted by keys (natural order or comparator)
Performance	O(1) for put/get/remove	O(1) with insertion order	O(log n) for put/get/remove (since it's a balanced tree)
Null Handling	Allows one null key, multiple null values	Same as `HashMap`	No null keys (null values allowed)

5. Queue Implementations

Feature	**PriorityQueue**	**ArrayDeque**	**LinkedList (Queue)**
Ordering	Elements ordered by natural ordering or by a comparator	Deque (double-ended queue), can act as both stack and queue	Implements both `Queue` and `Deque`, acts as a FIFO queue
Use Case	Best for scheduling tasks based on priority	Ideal for fast, double-ended operations (queue/stack hybrid)	Useful for queue-based operations

20.8 Other Important Differences:

1. Synchronized vs. Non-Synchronized Collections

- **Synchronized collections** are thread-safe but generally slower.
  - Example: `Vector`, `Hashtable`, `Collections.synchronizedList(new ArrayList())`
- **Non-synchronized collections** are faster but not thread-safe.
  - Example: `ArrayList`, `HashSet`, `HashMap`

2. Sorted vs. Unsorted Collections

- **Sorted collections** (e.g., `TreeSet`, `TreeMap`) automatically arrange elements in natural or custom order.
- **Unsorted collections** (e.g., `HashSet`, `HashMap`) do not maintain any order.

3. Navigable Collections

- **Navigable** collections (`TreeSet`, `TreeMap`) provide methods like `ceiling()`, `floor()`, `higher()`, and `lower()` to navigate through sorted collections.

20.9 Conclusion:

- The **Java Collections Framework** is like a **toolbox** that provides various data structures and methods for storing, managing, and retrieving collections of objects efficiently. It offers a wide range of options to suit different needs, making data manipulation easier and more organized.

21. Java Generics

21.1 What is it?

- **Java Generics** is a feature that allows you to define classes, interfaces, and methods with a placeholder for types, enabling you to create code that is type-safe and reusable without sacrificing performance.

21.2 Analogy: Universal Toolbox

Imagine a **universal toolbox** that can hold various types of tools—wrenches, screwdrivers, hammers, etc. This toolbox is designed in such a way that you can specify what type of tool you want to store in it without needing to create a different toolbox for each type.

21.3 Key Points:

1. **Type Safety:**
   - Just like a universal toolbox that ensures you only store compatible tools (e.g., no spoons in a toolbox for wrenches), generics allow you to create collections that ensure type safety. This prevents runtime errors caused by mixing different types.
2. **Code Reusability:**
   - A universal toolbox can be used for various types of tools, making it versatile. Similarly, generics enable you to write code that can work with any object type, making your code more flexible and reusable.
3. **Avoiding Casts:**
   - With a regular toolbox, you might need to check and cast tools when retrieving them. A universal toolbox designed for a specific type eliminates the need for this extra step. Generics in Java help avoid explicit casting, which reduces errors and simplifies code.

21.4 Example in Java:

```
import java.util.ArrayList;


public class GenericToolboxExample {
    // A generic class that can hold any type of tool
    static class Toolbox<T> {
        private ArrayList<T> tools = new ArrayList<>();


        public void addTool(T tool) {
            tools.add(tool);
        }
        public T getTool(int index) {
            return tools.get(index);
        }
    }

    public static void main(String[] args) {
        // Creating a toolbox for wrenches
        Toolbox<String> wrenchBox = new Toolbox<>();
        wrenchBox.addTool("Adjustable Wrench");
        wrenchBox.addTool("Pipe Wrench");

        // Creating a toolbox for screwdrivers
        Toolbox<String> screwdriverBox = new Toolbox<>();
        screwdriverBox.addTool("Flathead Screwdriver");
        screwdriverBox.addTool("Phillips Screwdriver");

        // Retrieving tools
        System.out.println("Wrenches: " + wrenchBox.getTool(0) + ", " + wrenchBox.getTool(1));
        System.out.println("Screwdrivers: " + screwdriverBox.getTool(0) + ", " +
screwdriverBox.getTool(1));
    }
}
```

21.5 Key Features:

1. **Generic Class:** The `Toolbox<T>` class is defined with a type parameter `<T>`, allowing it to hold any type of tool. This is similar to a universal toolbox that can hold different kinds of tools.
2. **Type Parameter:** When creating instances of `Toolbox`, you specify the type of tool it will hold (e.g., `Toolbox<String>` for string tools), ensuring type safety.
3. **No Casting Needed:** When retrieving tools from the toolbox, there's no need for casting, which simplifies the code and reduces the chance of errors.

21.6 Conclusion:

- **Java Generics** function like a **universal toolbox** that can hold any type of object safely and efficiently. By providing type safety, enhancing code reusability, and eliminating the need for casting, generics make your Java programs cleaner and more robust.

22. Java Strings

22.1 What is a String in Java?

In Java, a **String** is a **sequence of characters**. It is one of the most commonly used data types in programming to handle text. Strings in Java are **immutable**, which means once a string is created, its value cannot be changed.

22.2 Analogy: String as a Necklace

Imagine a **necklace** where each **bead** represents a character, and the entire necklace is a **string** of beads (characters).

- The **beads** are characters (`a`, `b`, `c`, etc.).
- The **necklace** is the **String** that holds all the beads together in a specific sequence.

22.3 Key Characteristics of Strings:

1. **Immutability**: Once a string (necklace) is created, you cannot change the beads (characters) in that string. If you try to modify it, Java creates a new string instead of changing the existing one.
   - **Analogy**: Imagine you have a necklace with specific beads in a certain order. If you want to change the beads, you don't change the beads on the existing necklace; you create a **new necklace** with the updated beads.

22.3.1 Creating a String:

There are two main ways to create a string in Java:

1. **Using string literals**:

String str = "Hello";

Analogy: It's like picking a pre-made necklace from a store.

2. **Using the `new` keyword**:

String str = new String("Hello");

Analogy: It's like making a necklace yourself by assembling beads.

22.4 Important String Methods:

Java provides various methods to manipulate and work with strings. Here are a few common ones:

1. **`length()`**: Finds the length of the string (i.e., how many characters it has).
   - **Analogy**: It counts the number of beads on the necklace.

```
String str = "Hello";

int len = str.length();  // Returns 5
```

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

Analogy: It's like pointing to a specific bead on the necklace based on its position.

String str = "Hello";

char ch = str.charAt(1); // Returns 'e'

3. `substring(int start, int end)`: Extracts a part of the string (a portion of the necklace).

Analogy: Imagine cutting a section of the necklace to create a smaller necklace with some beads from the original one.

String str = "Hello";

String sub = str.substring(1, 4); // Returns "ell"

4. `toUpperCase()` **and** `toLowerCase()`: Converts the string to upper or lower case.

Analogy: Imagine changing the **color** of all the beads to either bright (uppercase) or dim (lowercase).

String str = "Hello";

String upper = str.toUpperCase(); // Returns "HELLO"

String lower = str.toLowerCase(); // Returns "hello"

5. `equals(String anotherString)`: Checks if two strings are exactly the same.

Analogy: You compare two necklaces bead by bead to see if they are identical.

String str1 = "Hello";

String str2 = "Hello";

boolean isEqual = str1.equals(str2); // Returns true

6. `concat(String anotherString)`: Concatenates (joins) two strings together.

Analogy: It’s like connecting two necklaces to make a longer one.

String str1 = "Hello";

String str2 = "World";

String result = str1.concat(str2); // Returns "HelloWorld"

22.5 Immutability Explained:

Since strings in Java are immutable, any operation that seems like it modifies the string actually creates a **new string**.

```
String str = "Hello";

str = str.concat(" World");

System.out.println(str);  // Outputs "Hello World"
```

Here, the original string `"Hello"` is not changed. Instead, a new string `"Hello World"` is created, and the variable `str` now refers to this new string.

Analogy: If you have a necklace and want to add new beads, you can't modify the old necklace. Instead, you create a new necklace with the old beads plus the new ones.

22.6 String Pool in Java:

Java uses a **String Pool** to save memory by reusing strings.

22.6.1 How it works:

- When you create a string using a **literal** (`"Hello"`), Java checks if a string with the same value already exists in the string pool. If it does, it reuses the existing string instead of creating a new one.
- However, when you use the `new` keyword (`new String("Hello")`), Java creates a new string even if an identical one exists in the pool.
- **Analogy**: The string pool is like a **jewelry box** that holds necklaces. If you pick a necklace (string literal) from the box, it reuses the existing necklace. If you make a new necklace (using `new`), it doesn't care what's in the box—it creates a new one.

22.6.2 Example:

```
String str1 = "Hello";  // Stored in the string pool

String str2 = "Hello";  // Reuses the same string from the pool


String str3 = new String("Hello");  // Creates a new object outside the pool
```

22.7 Summary of Java Strings:

- A **string** is a sequence of characters and is immutable (cannot be changed once created).
- **Analogy**: Strings are like **necklaces**, with each character being a bead.
- Java provides several methods for manipulating strings, like `length()`, `substring()`, `toUpperCase()`, etc.
- Strings created using **literals** are stored in a **string pool** for memory efficiency, while those created with the `new` keyword are separate objects.

23. String Builder

In Java, **StringBuilder** is a mutable sequence of characters, which means you can modify it (change its contents) without creating new objects, unlike the immutable `String` class. **StringBuilder** is more efficient when dealing with a large number of string manipulations, such as appending, inserting, or deleting characters.

23.1 Key Features of `StringBuilder`

- **Mutable**: You can modify the contents of a `StringBuilder` without creating new objects.
- **Efficient**: It is more efficient than `String` when you have to repeatedly modify the same string, especially in loops.
- **No Synchronization**: Unlike `StringBuffer`, `StringBuilder` is not synchronized, making it faster in single-threaded contexts.

23.2 Basic Operations with `StringBuilder`

Here are some of the key methods and how they work:

1. Creating a StringBuilder

You can create a `StringBuilder` using different constructors:

StringBuilder sb = new StringBuilder(); // Empty StringBuilder

StringBuilder sbWithString = new StringBuilder("Hello"); // StringBuilder with an initial string

2. append()

This method adds text to the end of the `StringBuilder` sequence.

- **Example**:

StringBuilder sb = new StringBuilder("Hello");

sb.append(", World!"); // sb will be "Hello, World!"

3. insert(int offset, String str)

Inserts a string at a specified position in the `StringBuilder`.

StringBuilder sb = new StringBuilder("Hello!");

sb.insert(5, " Java"); // sb will be "Hello Java!"

4. delete(int start, int end)

Deletes the characters in the `StringBuilder` from the `start` index to the `end` index (exclusive).

- **Example**

```
StringBuilder sb = new StringBuilder("Hello, World!");
sb.delete(5, 7);  // sb will be "HelloWorld!"
```

5. replace(int start, int end, String str)

Replaces the characters between the `start` and `end` index with the specified string.

- Example

```
StringBuilder sb = new StringBuilder("Hello, World!");
sb.replace(7, 12, "Java");  // sb will be "Hello, Java!"
```

6. reverse()

Reverses the entire sequence of characters in the `StringBuilder`.

- **Example:**

```
StringBuilder sb = new StringBuilder("Hello");
sb.reverse();  // sb will be "olleH"
```

7. charAt(int index)

Returns the character at the specified index.

- **Example:**

```
StringBuilder sb = new StringBuilder("Hello");
char ch = sb.charAt(1);  // ch will be 'e'
```

8. toString()

Converts the `StringBuilder` to a `String`.

- **Example:**

```
StringBuilder sb = new StringBuilder("Hello");
String result = sb.toString();  // result will be "Hello"
```

9. setCharAt(int index, char ch)

Modifies the character at the given index.

- **Example:**

```
StringBuilder sb = new StringBuilder("Hello");

sb.setCharAt(1, 'a');  // sb will be "Hallo"
```

10. capacity()

Returns the current capacity of the `StringBuilder`. The capacity grows as the `StringBuilder` needs more space.

- **Example:**

```
StringBuilder sb = new StringBuilder();

int capacity = sb.capacity();  // capacity will be the default 16 or more if the string is larger
```

23.3 When to Use `StringBuilder`

- **String concatenation in loops**: If you are appending strings multiple times (e.g., in a loop), using `StringBuilder` is more efficient

```
StringBuilder sb = new StringBuilder();

for (int i = 0; i < 5; i++) {

    sb.append(i).append(" ");

}

System.out.println(sb.toString());  // Outputs: 0 1 2 3 4
```

23.4 Performance Comparison

- **String** is immutable, so every modification creates a new object. This is inefficient when you have many string operations.
- **StringBuilder** is mutable, so it avoids creating unnecessary objects, leading to better performance when modifying strings frequently.

23.5 String vs. StringBuilder

Aspect	String	StringBuilder
Mutability	Immutable (cannot change once created)	Mutable (can modify without new objects)
Efficiency	Slower for repeated modifications	Faster for repeated modifications
Thread-Safety	Thread-safe (safe in multithreaded apps)	Not thread-safe (use in single threads)
Use Case	When the value should remain constant	When string modifications are frequent

23.6 Summary

- `StringBuilder` is used when you need to modify strings frequently, such as appending, inserting, or deleting characters.
- It provides better performance compared to `String` when dealing with multiple string manipulations.

24. Differences between StringBuilder and StringBuffer

Aspect	StringBuilder	StringBuffer
Thread-Safety	Not thread-safe (not synchronized)	Thread-safe (synchronized methods)
Performance	Faster due to no synchronization	Slower because of synchronization
Use Case	Single-threaded applications (better speed)	Multi-threaded applications (safety first)

24.1 Explanation

1. Thread-Safety:

- **StringBuilder** is **not synchronized**, which means it does not guarantee safe usage in a **multi-threaded** environment. If multiple threads access a `StringBuilder` at the same time, you might encounter data corruption or unexpected behavior.
- **StringBuffer**, on the other hand, is **synchronized** and **thread-safe.** All of its methods are synchronized, which means only one thread can access the `StringBuffer` object at a time. This ensures that there are no conflicts between threads, making `StringBuffer` a safer choice in **multi-threaded** environments.

2. Performance:

- **StringBuilder** is generally **faster** because it doesn't have the overhead of synchronization. Since it's not thread-safe, it can perform string modifications more quickly in **single-threaded** applications.
- **StringBuffer**, being synchronized, is **slower** due to the extra steps it takes to ensure thread safety. Every method in `StringBuffer` has to lock the object, perform the operation, and then release the lock, which makes it less efficient in a single-threaded context.

3. Use Case:

- Use **`StringBuilder`** when you're working in a **single-threaded environment** and need to modify strings frequently, like appending, inserting, or deleting characters. It provides better performance because it avoids the cost of synchronization.
- Use **`StringBuffer`** when you're working in a **multi-threaded environment**, where multiple threads may access and modify the same string. Its thread safety ensures that the string operations are performed without issues.

24.2 Methods in `StringBuilder` and `StringBuffer`

Both `StringBuilder` and `StringBuffer` share almost the same API and methods, including:

- `append()`
- `insert()`
- `delete()`
- `replace()`
- `reverse()`
- `toString()`

These methods work the same way in both classes, but the performance and thread safety aspects differ.

24.3 Example: StringBuilder vs StringBuffer

```
// StringBuilder example (Single-threaded)

StringBuilder sb = new StringBuilder("Hello");

sb.append(", World!");

System.out.println(sb);  // Outputs: Hello, World!


// StringBuffer example (Multi-threaded)

StringBuffer sbuf = new StringBuffer("Hello");

sbuf.append(", World!");

System.out.println(sbuf);  // Outputs: Hello, World!
```

In both cases, the output is the same, but if you have multiple threads accessing `sbuf`, `StringBuffer` ensures the correct behavior, whereas `StringBuilder` might not.

24.4 When to Use `StringBuilder` or `StringBuffer`

- **Use `StringBuilder`** when:
  - You are working in a **single-threaded** application.
  - You need to optimize performance for frequent string modifications.
- **Use `StringBuffer`** when:
  - You are working in a **multi-threaded** environment where thread safety is required.

24.5 Summary

- `StringBuilder` is faster and more efficient for single-threaded applications because it is not synchronized.
- `StringBuffer` is thread-safe and should be used in multi-threaded environments, but it is slower due to synchronization overhead.

25. Java Streams

25.1 What is it?

- **Java Streams** are a part of the Java Collections Framework that allow you to process sequences of elements (like collections of data) in a functional style. They enable developers to perform operations such as filtering, mapping, and reducing in a clean and efficient way.

25.2 Analogy: River

Think of **Java Streams** as a **river** flowing through a landscape. The river carries water (data) and can have various streams (tributaries) that represent different operations you can perform on that data. Just as the river can split into smaller streams to handle specific tasks, Java Streams can be transformed and processed through a series of operations.

25.3 Key Points:

1. **Flowing Data**:
   - Just like a river carries water smoothly from one place to another, Java Streams allow you to process data in a sequence. You can take a collection of items and stream them through a series of operations.
2. **Transformations**:
   - The river can split into different tributaries where it can be filtered or diverted (e.g., a stream for fish, another for water plants). Similarly, Java Streams can be transformed through operations like `filter()`, `map()`, and `reduce()`, allowing you to shape the data as needed.
3. **Lazy Evaluation**:
   - The flow of water in the river is continuous, but it only does work when necessary (like eroding banks or carrying debris). Java Streams use lazy evaluation, meaning operations are not performed until a terminal operation (like `collect()` or `forEach()`) is invoked, which helps optimize performance.

25.4 Example in Java:

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;
```

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

        // Using streams to filter, square, and collect results
        List<Integer> evenSquares = numbers.stream() // Create a stream from the list
                .filter(n -> n % 2 == 0) // Filter even numbers
                .map(n -> n * n) // Square the numbers
                .collect(Collectors.toList()); // Collect the results into a list

        // Displaying the results
        System.out.println("Even Squares: " + evenSquares);
    }
}
```

25.5 Key Features:

1. **Stream Creation:** The `stream()` method converts a collection into a stream, allowing you to begin processing the data.
2. **Intermediate Operations:** Methods like `filter()` and `map()` are intermediate operations that transform the data without modifying the original collection.
3. **Terminal Operation:** The `collect()` method is a terminal operation that triggers the processing of the stream and collects the results into a new list.

25.6 Conclusion:

- **Java Streams** are like a **river** flowing through a landscape, enabling the smooth and efficient processing of data. They allow for a functional approach to data manipulation, with transformations and lazy evaluation that optimize performance.

26. Java Lambdas

26.1 What is it?

- **Java Lambdas** are a feature introduced in Java 8 that allows you to express instances of single-method interfaces (functional interfaces) in a concise and readable way. They enable you to write anonymous functions that can be treated as first-class citizens in Java, making it easier to implement functional programming concepts.

26.2 Analogy: Recipe Card

Think of a **lambda expression** as a **recipe card** for cooking. The recipe card contains a list of instructions (the logic) that you can follow to create a dish (the action). Just like you can use a recipe card without formally naming the dish or writing out the entire process every time, a lambda allows you to define a method inline without needing a separate class.

26.3 Key Points:

1. **Concise Definition**:
   - A recipe card provides a quick and straightforward way to prepare a dish without the need for an elaborate cookbook. Similarly, a lambda expression provides a shorthand way to define a function without creating a separate method or class.
2. **Parameters and Body**:
   - Just as a recipe card may include ingredients (parameters) and cooking steps (the body), a lambda expression takes parameters and contains the code that defines what to do with those parameters.
3. **Anonymous**:
   - A recipe card can be used without needing to know who wrote it or its history. In the same way, lambdas are anonymous functions that can be passed around without needing a formal name or class definition.

26.4 Syntax of a Lambda Expression:

The syntax of a lambda expression follows this format:

```
java
Copy code
(parameters) -> { // code to execute }
```

- **Example:** `(int x, int y) -> x + y` represents a function that takes two integers and returns their sum.

26.5 Example in Java:

import java.util.Arrays;

import java.util.List;

```
public class LambdaExample {
    public static void main(String[] args) {
        // Creating a list of names
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

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

        // Using a lambda expression to filter names that start with 'A'
        List<String> filteredNames = names.stream()
            .filter(name -> name.startsWith("A"))
            .collect(Collectors.toList());

        // Displaying the filtered names
        System.out.println("Names starting with 'A': " + filteredNames);
    }
}
```

26.6 Key Features:

1. **Functional Interface**: The `forEach` method and the `filter` method expect a functional interface (like `Consumer` or `Predicate`), which is implemented by the lambda expressions.
2. **Inline Logic**: The lambda expression provides the implementation of the method directly, making the code more concise and readable.
3. **Readable Code**: By using lambdas, the code becomes easier to read and understand, resembling the steps on a recipe card rather than cluttered class definitions.

26.7 Conclusion:

- **Java Lambdas** are like **recipe cards** that provide a simple and concise way to define and implement functional behavior in Java. They enhance the language's expressiveness and enable developers to write cleaner, more maintainable code.