This document provides a comprehensive study guide for Java programming, covering fundamental concepts, object-oriented principles, advanced features, data structures, I/O, multithreading, database access, and modern Java (Java 8+).

---

## Java Language and its Features

Java is a high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let application developers *write once, run anywhere* (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

**Key Features of Java:**

- **Object-Oriented:** Java is a pure object-oriented language (except for primitive data types). It supports concepts like encapsulation, inheritance, polymorphism, and abstraction.
- **Platform Independent:** Achieved through the Java Virtual Machine (JVM). Compiled Java bytecode (`.class` files) can run on any platform with a compatible JVM.
- **Simple:** Designed to be easy to learn and use, especially for programmers familiar with C++. It removes complex features like pointers and explicit memory management.
- **Secure:** Features like bytecode verifier, security manager, and absence of pointers contribute to its security. Java applications run in a sandbox environment.
- **Robust:** Strong memory management, automatic garbage collection, and exception handling mechanisms make Java applications robust and reliable.
- **Multithreaded:** Java has built-in support for multithreaded programming, allowing concurrent execution of multiple parts of a program.
- **High Performance:** While interpreted initially, modern JVMs use Just-In-Time (JIT) compilers to compile bytecode into native machine code at runtime, leading to high performance.
- **Distributed:** Designed for distributed computing, allowing applications to work across networks.
- **Dynamic:** Capable of adapting to an evolving environment. Classes can be loaded on demand.

### JDK, JRE & JVM

These three components are fundamental to the Java ecosystem.

- **JVM (Java Virtual Machine):**
  - **What it is:** An abstract machine that provides a runtime environment in which Java bytecode can be executed. It is the core of Java's platform independence.
  - **Role:**
    1. Loads `.class` files.
    2. Verifies bytecode.
    3. Executes bytecode.
    4. Provides a runtime environment for Java applications.
    5. Manages memory (garbage collection).
  - **Platform Specific:** While Java code is platform-independent, the JVM itself is platform-dependent. A different JVM implementation is needed for each operating system.
- **JRE (Java Runtime Environment):**

    - **What it is:** A software package that provides the minimum requirements for running a Java application. It includes the JVM, core Java libraries, and other components necessary for the execution of Java programs.
    - **Role:** To run compiled Java applications. It does NOT contain tools for developing Java applications (like compilers).
    - **Components:** JVM + Java core class libraries (`rt.jar`).
- **JDK (Java Development Kit):**
    - **What it is:** A software development environment used for developing Java applications and applets. It includes the JRE, along with development tools (compiler, debugger, archiver, etc.).
    - **Role:** For Java developers to write, compile, and run Java programs.
    - **Components:** JRE + development tools (`javac` for compilation, `java` for execution, `jdb` for debugging, `javap` for disassembler, `jar` for archiving).

### JVM Architecture Overview

The JVM's architecture describes how it executes Java programs. It's broadly divided into three main subsystems:

1. **Classloader Subsystem:**
    - **Purpose:** Loads `.class` files from the disk into memory.
    - **Steps:**
        - **Loading:** Finds and loads class files (using `Bootstrap Classloader`, `Extension Classloader`, `Application Classloader`).
        - **Linking:**
            - **Verification:** Ensures the bytecode is valid and safe.
            - **Preparation:** Allocates memory for static variables and initializes them to default values.
            - **Resolution:** Replaces symbolic references in the bytecode with direct references to actual memory addresses.
        - **Initialization:** Executes the static initializers (`static {}` blocks and static variable assignments) of the class.
2. **Runtime Data Areas (Memory Areas):**
    - **Purpose:** Memory allocated by the JVM to run the program.
    - **Areas:**
        - **Method Area:** Stores class-level data (metadata, static variables, constant pool, method code). It's shared among all threads.
        - **Heap Area:** Where all objects and their corresponding instance variables and arrays are allocated. It's shared among all threads. Garbage Collection primarily operates on the Heap.
        - **Stack Area (JVM Stacks):** Each thread has its own private JVM stack. Stores method call frames (local variables, operand stack, frame data).
        - **PC Registers (Program Counter Registers):** Each thread has its own private PC register. Stores the address of the next instruction to be executed.
        - **Native Method Stacks:** Each thread has its own private native method stack. Stores information for native methods (methods written in languages other than Java, like C/C++).
3. **Execution Engine:**

- **Purpose:** Executes the bytecode.
- **Components:**
  - **Interpreter:** Reads and executes bytecode instructions one by one.
  - **JIT (Just-In-Time) Compiler:** Improves performance by compiling frequently executed bytecode (hotspots) into native machine code. It caches the compiled code for reuse.
  - **Garbage Collector (GC):** Automatically reclaims memory occupied by objects that are no longer referenced by the program.
  - **Native Method Interface (JNI):** Allows Java code to interact with native applications and libraries (e.g., C/C++ code).
  - **Native Method Libraries:** Libraries (written in native languages) required by the Execution Engine.

## Data types, Variables, Constants, Operators, Control Statements

**Data Types:** Define the type of data a variable can hold.

- **Primitive Data Types:** Basic data types that are not objects.
  - `byte` (1 byte, -128 to 127)
  - `short` (2 bytes)
  - `int` (4 bytes)
  - `long` (8 bytes)
  - `float` (4 bytes, single-precision floating-point)
  - `double` (8 bytes, double-precision floating-point - default for decimals)
  - `boolean` (1 bit, `true` or `false`)
  - `char` (2 bytes, Unicode character)
- **Reference Data Types:** Refer to objects. They store the memory address (reference) where the object is located.
  - `String`
  - Arrays
  - Classes
  - Interfaces

**Variables:** Named memory locations that store data.

- **Declaration:** `dataType variableName;`
- **Initialization:** `dataType variableName = value;`
- **Types of Variables:**
  - **Local Variables:** Declared inside methods, constructors, or blocks. Scope is limited to the block. No default value.
  - **Instance Variables (Non-static fields):** Declared inside a class but outside any method, constructor, or block. Each object instance has its own copy. Default values (0, false, null).
  - **Static Variables (Class variables):** Declared with `static` keyword. Belongs to the class, not any specific object. One copy shared by all instances. Default values.

**Constants:** Variables whose values cannot be changed after initialization.

- Declared using the `final` keyword.
- Often declared as `public static final` for global constants.

```java
final double PI = 3.14159;
public static final int MAX_USERS = 100;
```

**Operators:** Symbols that perform operations on variables and values.

- **Arithmetic:** `+`, `-`, `*`, `/`, `%`
- **Assignment:** `=`, `+=`, `-=`, etc.
- **Relational (Comparison):** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Logical:** `&&` (AND), `||` (OR), `!` (NOT)
- **Bitwise:** `&`, `|`, `^`, `~`, `<<`, `>>`, `>>>`
- **Unary:** `+`, `-`, `++`, `--`, `!`
- **Ternary (Conditional):** `condition ? expr1 : expr2`

**Control Statements:** Statements that control the flow of execution in a program.

- **Conditional Statements:**
  - `if-else if-else`: Executes different blocks of code based on conditions.
  - `switch`: Selects one of many code blocks to be executed.
- **Looping Statements:**
  - `for` loop: Iterates a block of code a specified number of times.
  - `while` loop: Repeats a block of code as long as a condition is true.
  - `do-while` loop: Executes a block of code at least once, then repeats as long as a condition is true.
  - `for-each` loop (Enhanced for loop): Iterates over elements of arrays or collections.
- **Jump Statements:**
  - `break`: Terminates the innermost loop or `switch` statement.
  - `continue`: Skips the current iteration of a loop and continues with the next.
  - `return`: Exits a method and optionally returns a value.

---

## Discussion on Object-Oriented Concepts

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data (attributes/properties) and code (methods/behaviors).

### Classes and Objects

- **Class:**
  - **Blueprint/Template:** A blueprint or template for creating objects. It defines the common attributes and behaviors that objects of that class will have.
  - **Logical Entity:** A class is a logical entity; it does not occupy memory when defined.
  - **Example:** A `Car` class defines properties like `color`, `make`, `model` and behaviors like `start()`, `stop()`.
- **Object:**
  - **Instance:** An instance of a class. When you create an object, you allocate memory for it based on the class's blueprint.
  - **Real-World Entity:** Represents a real-world entity.

- **Runtime Entity:** An object is a runtime entity that occupies memory.
- **Example:** `myCar = new Car("Red", "Toyota", "Camry");` creates an object `myCar` based on the `Car` class.

## Access Specifiers, Namespaces and Tokens

- **Access Specifiers (Access Modifiers):** Keywords that set the accessibility (visibility) of classes, constructors, methods, and data members.
  - **`public`:** Accessible from anywhere (within the class, outside the class, within the package, outside the package).
  - **`protected`:** Accessible within the same package and by subclasses (even if in different packages).
  - **`default` (no keyword):** Accessible only within the same package.
  - **`private`:** Accessible only within the same class. (Highest restriction)
- **Namespaces (Packages in Java):**
  - **Purpose:** A mechanism to organize classes and interfaces into logical groups. Prevents naming conflicts between classes with the same name from different developers/libraries.
  - **Convention:** Follows a reverse domain name convention (e.g., `com.example.myapp.utilities`).
  - **`import` statement:** Used to bring classes from other packages into the current scope.
- **Tokens:** The smallest individual units of a program.
  - **Keywords:** Reserved words with special meaning (e.g., `public`, `class`, `if`, `while`).
  - **Identifiers:** Names given to variables, methods, classes, etc. (e.g., `myVariable`, `calculateSum`).
  - **Literals:** Constant values (e.g., `10`, `"hello"`, `3.14`, `true`).
  - **Operators:** Symbols that perform operations (`+`, `=`, `==`).
  - **Separators:** Punctuation marks (e.g., `;`, `{`, `}`, `(`, `)`).

## Constructors and Destructors

- **Constructors:**
  - **Purpose:** Special methods used to initialize the state of an object when it is created (instantiated).
  - **Name:** Must have the same name as the class.
  - **Return Type:** No return type (not even `void`).
  - **Invocation:** Called automatically when `new` operator is used.
  - **Types:**
    - **Default Constructor:** Provided by the Java compiler if no explicit constructor is defined. Initializes instance variables to default values.
    - **No-argument Constructor:** A constructor with no parameters explicitly defined by the programmer.
    - **Parameterized Constructor:** A constructor that takes arguments to initialize instance variables with specific values.
  - **Constructor Overloading:** A class can have multiple constructors with different parameter lists.
- **Destructors (in Java):**
  - **Concept:** In languages like C++, destructors are special methods called when an object is destroyed to release resources.

  - **Java's Approach:** Java does **NOT** have destructors in the traditional sense. Memory deallocation (garbage collection) is handled automatically by the JVM.
  - **`finalize()` method:** While Java has a `finalize()` method (in the `Object` class), it is **deprecated** and generally **not recommended** for cleanup. Its execution is not guaranteed, can be unpredictable, and may impact performance. For resource cleanup, use `try-with-resources` or explicit `close()` methods.

### Classes in Java

As discussed above, classes are blueprints. In Java, a class definition includes:

- **Access Specifier:** (`public`, `default`)
- **`class` keyword:**
- **Class Name:**
- **Fields (Instance Data/Variables):** Attributes of the objects.
- **Constructors:** For object initialization.
- **Methods:** Behaviors/actions of the objects.

**Example:**

```
public class Dog { // Class definition
    // Instance Data (Fields)
    String name;
    String breed;
    int age;

    // Constructors
    public Dog(String name, String breed, int age) { // Parameterized Constructor
        this.name = name;
        this.breed = breed;
        this.age = age;
    }

    public Dog(String name, String breed) { // Overloaded Constructor
        this(name, breed, 0); // Calls another constructor (constructor chaining)
    }

    // Instance Methods
    public void bark() {
        System.out.println(name + " says Woof!");
    }

    public void displayInfo() {
        System.out.println("Name: " + name + ", Breed: " + breed + ", Age: " +
age);
    }
}
```

### Constructors, instance data and methods, the `new` operator

- `new` **operator:** Used to create an instance (object) of a class. When `new` is used, it:
    1. Allocates memory for the new object on the heap.
    2. Initializes instance variables to their default values (0 for numbers, `false` for boolean, `null` for objects).
    3. Invokes the specified constructor.
    4. Returns a reference (memory address) to the newly created object.

**Example (using the `Dog` class):**

```java
public class DogTester {
    public static void main(String[] args) {
        // Creating an object using the 'new' operator and a constructor
        Dog myDog = new Dog("Buddy", "Golden Retriever", 3); // Calls the parameterized constructor

        // Accessing instance data (fields)
        System.out.println("My dog's name is: " + myDog.name);
        System.out.println("My dog's breed is: " + myDog.breed);

        // Calling instance methods
        myDog.bark();
        myDog.displayInfo();

        Dog anotherDog = new Dog("Lucy", "Labrador"); // Calls the overloaded constructor
        anotherDog.displayInfo();
    }
}
```

---

## Static variables and methods

The `static` keyword in Java is used to create class-level members, meaning they belong to the class itself rather than to any specific instance (object) of the class.

- **Static Variables (Class Variables):**
    - **Ownership:** Belongs to the class, not to objects.
    - **Memory:** Stored in the Method Area of the JVM. Only one copy exists for the entire class, regardless of how many objects are created (or even if no objects are created).
    - **Initialization:** Initialized only once, when the class is loaded into memory.
    - **Access:** Accessed directly using the class name (`ClassName.staticVariable`) or through an object reference (though class name is preferred for clarity).
    - **Use Cases:** Constants (`public static final`), shared counters, configuration data common to all instances.
- **Static Methods (Class Methods):**
    - **Ownership:** Belongs to the class.
    - **Invocation:** Called directly using the class name (`ClassName.staticMethod()`).

- **Access Restrictions:** Static methods can *only* access other static members (variables and methods) directly. They *cannot* access instance variables or instance methods because they don't operate on a specific object instance.
- **`this` and `super`:** Cannot use `this` or `super` keywords inside static methods.
- **Use Cases:** Utility methods that don't need object state (e.g., `Math.sqrt()`), factory methods, main method.

### Accessing static variables and methods of different class

You access static members of another class directly using the class name.

**Example:**

```java
class Calculator {
    public static final double PI = 3.14159; // Static variable (constant)
    private static int operationCount = 0; // Static variable (shared state)

    public static int add(int a, int b) { // Static method
        operationCount++;
        return a + b;
    }

    public static int subtract(int a, int b) { // Static method
        operationCount++;
        return a - b;
    }

    public static int getOperationCount() { // Static method to access static
variable
        return operationCount;
    }
}

public class StaticDemo {
    public static void main(String[] args) {
        // Accessing static variable directly using class name
        System.out.println("Value of PI: " + Calculator.PI);

        // Calling static methods directly using class name
        int sum = Calculator.add(10, 5);
        System.out.println("Sum: " + sum);

        int difference = Calculator.subtract(20, 7);
        System.out.println("Difference: " + difference);

        // Accessing shared static variable state
        System.out.println("Total operations: " + Calculator.getOperationCount());

        // Even without creating an object, static members are accessible
        // Calculator calc = new Calculator(); // No need to create object
        // System.out.println(calc.PI); // Still works, but not recommended
    }
}
```

### Introduction to reference data types

As mentioned, reference data types refer to objects.

- **What they store:** A memory address (reference) to the object's actual data on the heap.
- **Default value:** `null` (meaning no object is referenced).
- **Examples:** `String`, arrays, user-defined classes (`Dog`, `Calculator`), interfaces.

### Reference variables and methods

- **Reference Variable:** A variable that holds a reference to an object.

  ```
  Dog myDog; // myDog is a reference variable, initially null
  myDog = new Dog("Max", "German Shepherd", 2); // myDog now refers to a Dog
  object
  ```

- **Methods with Reference Parameters:** When an object is passed as an argument to a method, a copy of the *reference* is passed (pass-by-value of the reference). This means the method can modify the object's state (its instance variables) but cannot change which object the original reference variable points to.

**Example (Objects as parameters - discussed more in Session 10):**

```
class Wallet {
    int balance;
    Wallet(int b) { balance = b; }
    void addMoney(int amount) { balance += amount; }
}

public class ReferenceMethodDemo {
    public static void modifyWallet(Wallet walletRef, int amountToAdd) {
        walletRef.addMoney(amountToAdd); // Modifies the actual object
        walletRef = new Wallet(0); // This only changes the local walletRef, not
the original
    }

    public static void main(String[] args) {
        Wallet myWallet = new Wallet(100);
        System.out.println("Before: " + myWallet.balance); // 100

        modifyWallet(myWallet, 50); // Pass a copy of the reference
        System.out.println("After: " + myWallet.balance); // 150 (object state was
changed)

        // myWallet still refers to the original Wallet object, not the new one
created in modifyWallet
```

```
    }
}
```

### Difference between reference data types and primitive data types

| Feature | Primitive Data Types | Reference Data Types |
|---|---|---|
| **Storage** | Store the actual value directly. | Store the memory address (reference) of an object. |
| **Memory** | Stored on the stack (for local variables). | Object data stored on the heap. Reference on stack. |
| **Default Value** | Fixed (0, false, '\u0000'). | `null`. |
| **Size** | Fixed size (e.g., `int` is 4 bytes). | Fixed size (size of a memory address, usually 4 or 8 bytes), regardless of object size. |
| **Operations** | Direct arithmetic, comparison. | Operations on object state via methods, comparison of references. |
| **Examples** | `int`, `double`, `boolean`, `char`, etc. | `String`, arrays, custom class objects, interfaces. |

### Difference between reference variable and static variable

These concepts are orthogonal; a variable can be both `static` and a reference variable.

- **Reference Variable:** Describes *what kind of data it stores* (a reference to an object).
- **Static Variable:** Describes *how the variable behaves* (belongs to the class, one copy shared by all, stored in Method Area).

**Combinations:**

- **`int count;`**: Primitive, Instance variable.
- **`static int totalCount;`**: Primitive, Static variable.
- **`Dog myDog;`**: Reference, Instance variable.
- **`static Dog defaultDog;`**: Reference, Static variable. (One `Dog` object shared by all instances, or even if no instances exist.)

**Key takeaway:** A variable being `static` affects its lifetime and scope (class-level), while being a *reference* variable affects what kind of value it holds (an object's memory address).

---

## Abstraction, Data Hiding and Encapsulation

These are core principles of OOP, promoting modularity, security, and maintainability.

- **Abstraction:**

    - **Concept:** Showing only essential features/information while hiding complex implementation details. Focuses on "what" an object does rather than "how" it does it.
    - **Analogy:** A car's steering wheel and pedals (interface) abstract the complex engine and transmission (implementation).
    - **Implementation in Java:** Achieved through **abstract classes** and **interfaces**.
- **Data Hiding:**
    - **Concept:** Restricting direct access to an object's internal data (fields) from outside the class.
    - **Purpose:** Protects the integrity of data by preventing unauthorized or incorrect modifications.
    - **Implementation in Java:** Using the `private` access specifier for fields.
- **Encapsulation:**
    - **Concept:** Bundling data (fields) and the methods (behaviors) that operate on that data into a single unit (a class). It's about combining data hiding with public methods to control access.
    - **Analogy:** A capsule contains medication inside, and you interact with the capsule as a whole.
    - **Purpose:** Achieves data hiding and provides a controlled interface to interact with the object's state.
    - **Implementation in Java:**
        1. Declaring fields as `private`.
        2. Providing public `getter` and `setter` methods for controlled access to these private fields.

**Example of Encapsulation and Data Hiding:**

```java
public class Employee {
    private int id; // Data Hiding: id is private
    private String name;
    private double salary;

    public Employee(int id, String name, double salary) {
        this.id = id;
        this.name = name;
        // Apply validation during initialization
        if (salary >= 0) {
            this.salary = salary;
        } else {
            System.out.println("Salary cannot be negative. Setting to 0.");
            this.salary = 0;
        }
    }

    // Public Getter methods (read access) - Abstraction/Encapsulation
    public int getId() {
        return id; // Read-only access
    }

    public String getName() {
        return name;
    }

    public double getSalary() {
        return salary;
    }
    // Public Setter methods (write access with control) - Encapsulation
    public void setSalary(double salary) {
        if (salary >= 0) { // Encapsulation: Control logic for setting salary
            this.salary = salary;
        } else {
            System.out.println("Invalid salary. Must be non-negative.");
        }
    }

    public void displayDetails() {
        System.out.println("ID: " + id + ", Name: " + name + ", Salary: " +
salary);
    }
}

public class EmployeeDemo {
    public static void main(String[] args) {
        Employee emp = new Employee(101, "Alice", 50000);
        emp.displayDetails();
        // emp.salary = -100; // Compile-time error: salary is private (Data
Hiding)

        emp.setSalary(55000); // Controlled modification via setter
        emp.displayDetails();

        emp.setSalary(-500); // Validation logic in setter prevents invalid state
        emp.displayDetails();
    }
}
```

### Polymorphism - Runtime Polymorphism, Compile-time Polymorphism

**Polymorphism:** Means "many forms." It allows objects of different classes to be treated as objects of a common type (superclass or interface). The specific method implementation called depends on the actual type of the object at runtime.

- **Compile-time Polymorphism (Static Polymorphism/Method Overloading):**
    - **Concept:** Achieved through method overloading. Multiple methods in the same class have the same name but different parameter lists (different number, types, or order of parameters). The compiler determines which method to call at compile time based on the method signature.
    - **Example:**

    ```
    class Calculator {
        int add(int a, int b) { return a + b; }
        double add(double a, double b) { return a + b; } // Overloaded
    method
        int add(int a, int b, int c) { return a + b + c; } // Overloaded
    method
    }
    ```

```
Calculator calc = new Calculator();
calc.add(5, 10); // Calls int add(int, int)
calc.add(5.0, 10.0); // Calls double add(double, double)
```

- **Runtime Polymorphism (Dynamic Method Dispatch/Method Overriding):**
    - **Concept:** Achieved through method overriding. A subclass provides a specific implementation for a method that is already defined in its superclass. The actual method invoked is determined at runtime based on the actual object type, not the reference type.
    - **Requires:** Inheritance and method overriding.
    - **Example:**

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

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

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

public class RuntimePolymorphismDemo {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Animal reference, Dog object
        myAnimal.makeSound(); // Output: Dog barks (determined at
runtime)

        myAnimal = new Cat(); // Animal reference, Cat object
        myAnimal.makeSound(); // Output: Cat meows (determined at
runtime)
    }
}
```

## Inheritance and its types

**Inheritance:** A mechanism where one class (subclass/child class) acquires the properties and behaviors (fields and methods) of another class (superclass/parent class). It promotes code reusability and establishes an "is-a" relationship.

- **Keywords:** `extends` (for classes), `implements` (for interfaces).
- **Types of Inheritance in Java:** (Java only supports single inheritance for classes, but multiple inheritance of *types* via interfaces)

- **Single Inheritance:** A class inherits from only one superclass. (Most common and directly supported by `extends`).

  ```
  class A {}
  class B extends A {} // B inherits from A
  ```

- **Multilevel Inheritance:** A class inherits from a class, which in turn inherits from another class.

  ```
  class A {}
  class B extends A {}
  class C extends B {} // C inherits from B, B inherits from A
  ```

- **Hierarchical Inheritance:** Multiple subclasses inherit from a single superclass.

  ```
  class A {}
  class B extends A {}
  class C extends A {} // B and C both inherit from A
  ```

- **Hybrid Inheritance:** A combination of two or more types of inheritance. Java supports hybrid inheritance that involves single, multilevel, and hierarchical structures.
- **Multiple Inheritance (of implementation):** A class inheriting from multiple superclasses. **Java does NOT support multiple inheritance of *classes*** (to avoid the "Diamond Problem").
  - However, Java *does* support multiple inheritance of *type* via **interfaces** (a class can `implement` multiple interfaces).

### Templates

- **Java's equivalent to C++ Templates is Generics.**
- **Templates (in C++):** Allow writing generic programs that can operate on different data types without being rewritten for each type.
- **Generics (in Java - discussed later):** Provide similar functionality by allowing types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. This enables type-safe reusable code.

---

## Abstract class and abstract methods

**Abstract Class:** A class that cannot be instantiated directly (you cannot create an object of an abstract class). It acts as a blueprint for other classes, providing common functionality while also declaring abstract methods that must be implemented by its concrete subclasses.

- **Declaration:** Declared with the `abstract` keyword.
- **Instantiation:** Cannot be instantiated (`new AbstractClass()`).
- **Constructors:** Can have constructors, which are called by the constructors of its subclasses.

- **Methods:** Can have both abstract methods and concrete (non-abstract) methods.
- **Abstract Methods:**
    - **Declaration:** Declared with the `abstract` keyword and have no implementation (no method body, ends with a semicolon).
    - **Purpose:** Forces subclasses to provide their own implementation for these methods.
    - **Rule:** If a class contains any abstract method, the class *must* be declared abstract.
- **Inheritance:** Abstract classes must be subclassed by concrete classes that provide implementations for all inherited abstract methods. If a subclass does not implement all abstract methods, it too must be declared abstract.

**Example:**

```java
// Abstract Class
abstract class Shape {
    String color;

    // Constructor (called by subclasses)
    public Shape(String color) {
        this.color = color;
    }

    // Concrete method
    public void displayColor() {
        System.out.println("Color: " + color);
    }

    // Abstract methods (must be implemented by concrete subclasses)
    public abstract double getArea();
    public abstract void draw();
}

// Concrete Subclass
class Circle extends Shape {
    double radius;

    public Circle(String color, double radius) {
        super(color); // Call superclass constructor
        this.radius = radius;
    }

    @Override
    public double getArea() { // Implementation of abstract method
        return Math.PI * radius * radius;
    }

    @Override
    public void draw() { // Implementation of abstract method
        System.out.println("Drawing a Circle with radius " + radius + " and color
" + color);
    }
}
```

```
// Another Concrete Subclass
class Rectangle extends Shape {
    double width;
    double height;

    public Rectangle(String color, double width, double height) {
        super(color);
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle with width " + width + ", height "
+ height + " and color " + color);
    }
}

public class AbstractClassDemo {
    public static void main(String[] args) {
        // Shape s = new Shape("Green"); // Compile-time error: Cannot instantiate
abstract class Shape

        Shape circle = new Circle("Red", 5.0); // Polymorphism: Shape reference,
Circle object
        circle.displayColor();
        circle.draw();
        System.out.println("Circle Area: " + circle.getArea());

        Shape rectangle = new Rectangle("Blue", 4.0, 6.0); // Polymorphism: Shape
reference, Rectangle object
        rectangle.displayColor();
        rectangle.draw();
        System.out.println("Rectangle Area: " + rectangle.getArea());
    }
}
```

### Interface (implementing multiple interfaces)

**Interface:** A blueprint of a class. It contains only abstract methods (before Java 8) and public static final variables. Interfaces define a contract: any class that `implements` an interface must provide implementations for all the methods declared in that interface.

- **Declaration:** Declared with the `interface` keyword.
- **Instantiation:** Cannot be instantiated directly.
- **Fields:** All fields in an interface are implicitly `public static final`.

- **Methods:**
  - Before Java 8: All methods are implicitly `public abstract`.
  - Java 8 and above: Can also have `default` methods (with implementation) and `static` methods (with implementation).
- **Inheritance:**
  - A class `implements` one or more interfaces.
  - An interface `extends` one or more other interfaces (achieving multiple inheritance of type).
- **Purpose:**
  - Achieve full **abstraction**.
  - Support **multiple inheritance of type** (since a class can implement multiple interfaces).
  - Define **contracts** for behavior.
  - Achieve **loose coupling**.

**Example (implementing multiple interfaces):**

```java
// First Interface
interface Flyable {
    void fly(); // implicitly public abstract
    // default void glide() { System.out.println("Gliding..."); } // Java 8 default method
}

// Second Interface
interface Swimmable {
    void swim();
    // static void dive() { System.out.println("Diving..."); } // Java 8 static method
}

// Class implementing multiple interfaces
class Duck implements Flyable, Swimmable {
    @Override
    public void fly() { // Implementation for Flyable
        System.out.println("Duck is flying.");
    }

    @Override
    public void swim() { // Implementation for Swimmable
        System.out.println("Duck is swimming.");
    }
}

public class MultipleInterfaceDemo {
    public static void main(String[] args) {
        Duck myDuck = new Duck();
        myDuck.fly();
        myDuck.swim();

        // Using interface references (Polymorphism)
        Flyable bird = myDuck;
        bird.fly();
        // bird.swim(); // Compile-time error: Flyable reference doesn't know
about swim()

        Swimmable creature = myDuck;
        creature.swim();

        // Swimmable.dive(); // Call static method on interface
    }
}
```

---

## Final variables, final methods and final class

The `final` keyword in Java is used to restrict the modification of a variable, method, or class.

- `final` **variables:**
    - **Purpose:** Makes a variable a constant. Once initialized, its value cannot be changed.
    - **Initialization:** Must be initialized either at the time of declaration or within a constructor (for instance `final` variables) or a static block (for static `final` variables).
    - **Use Cases:** Defining constants (e.g., `public static final int MAX_VALUE = 100;`).

```
final int fixedValue = 10;
// fixedValue = 20; // Compile-time error

class MyClass {
    final String name; // Instance final variable, must be initialized in
constructor
    public MyClass(String name) { this.name = name; }
}
```

- `final` **methods:**
    - **Purpose:** Prevents a method from being overridden by subclasses.
    - **Use Cases:** When you want to ensure that a method's implementation remains unchanged across the inheritance hierarchy (e.g., core logic, utility methods).

```
class Parent {
    final void importantMethod() {
        System.out.println("This method cannot be overridden.");
    }
    void otherMethod() { System.out.println("Can be overridden."); }
}

class Child extends Parent {
    // @Override
    // void importantMethod() { /* ... */ } // Compile-time error: cannot
override final method
    @Override
    void otherMethod() { System.out.println("Child's other method."); }
}
```

- `final` **class:**
    - **Purpose:** Prevents a class from being subclassed.
    - **Use Cases:** For security reasons (e.g., `String` class is final to prevent malicious subclassing), to ensure immutability (e.g., all wrapper classes like `Integer`, `Double` are final), or when the design explicitly forbids extension.

```
final class ImmutablePoint {
    private final int x;
    private final int y;
    public ImmutablePoint(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
}

// class SubPoint extends ImmutablePoint { } // Compile-time error: cannot
inherit from final class
```

### Functional interface New interface features (Java 8 & above)

Java 8 introduced significant enhancements to interfaces.

- **Functional Interface:**
    - **Concept:** An interface that contains exactly one abstract method.
    - **Purpose:** Designed to be used with Lambda Expressions.
    - **Annotation:** `@FunctionalInterface` is a marker annotation (optional, but recommended) to indicate that an interface is intended to be a functional interface. The compiler enforces the single abstract method rule.
    - **Examples of Built-in Functional Interfaces:** `Runnable`, `Comparator`, `Callable`, `Consumer`, `Supplier`, `Function`, `Predicate`.

```
@FunctionalInterface
interface MyCalculator {
    int operate(int a, int b); // Single abstract method
    // int anotherMethod(); // Compile-time error if uncommented (more than
one abstract method)

    // Can have default methods
    default void printOperation() {
        System.out.println("Performing calculation.");
    }
}
```

- **Default Methods (Defender Methods):**
  - **Concept:** Methods with an implementation provided directly in the interface, using the `default` keyword.
  - **Purpose:** Allows adding new methods to existing interfaces without breaking backward compatibility for classes that already implement the interface.
  - **Use Cases:** Evolving APIs.
- **Static Methods in Interfaces:**
  - **Concept:** Methods with an implementation provided directly in the interface using the `static` keyword.
  - **Purpose:** Utility methods related to the interface.
  - **Access:** Can only be called using the interface name (e.g., `MyInterface.staticMethod()`). Cannot be overridden by implementing classes.

**Example of new interface features:**

```java
interface Drawable {
    void draw(); // Abstract method (single abstract method for functional interface)

    default void resize() { // Default method
        System.out.println("Resizing the drawable.");
    }

    static void showInstructions() { // Static method
        System.out.println("Instructions: Implement draw() method.");
    }
}

class Circle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle.");
    }
    // No need to implement resize()
}

public class InterfaceFeaturesDemo {
    public static void main(String[] args) {
        Circle circle = new Circle();
        circle.draw();
        circle.resize(); // Call default method

        Drawable.showInstructions(); // Call static method on interface
    }
}
```

## Arrays

An array is a data structure that stores a fixed-size sequential collection of elements of the same data type.

- **Declaration:** `dataType[] arrayName;` or `dataType arrayName[];`
- **Initialization:** `arrayName = new dataType[size];` or `arrayName = {value1, value2, ...};`
- **Index-based:** Elements are accessed using a zero-based index.
- **Fixed Size:** Once declared, the size of an array cannot be changed.
- **Homogeneous:** All elements must be of the same type.
- **Memory:** Arrays are objects in Java, stored on the heap.
- `length` **property:** Provides the number of elements in the array.

**Example:**

```java
// Declaration and initialization
int[] numbers = new int[5]; // Array of 5 integers, initialized to 0
numbers[0] = 10;
numbers[1] = 20;

// Declaration and initialization with values
String[] fruits = {"Apple", "Banana", "Cherry"};

// Accessing elements
System.out.println(fruits[0]); // Apple

// Iterating over an array
for (int i = 0; i < numbers.length; i++) {
    System.out.println("numbers[" + i + "] = " + numbers[i]);
}

// Enhanced for loop (for-each)
for (String fruit : fruits) {
    System.out.println(fruit);
}
```

- **Multidimensional Arrays:** Arrays of arrays. Common for matrices.

  ```java
  int[][] matrix = { {1, 2}, {3, 4, 5} };
  System.out.println(matrix[0][1]); // 2
  ```

## Enumerations

**Enumeration (Enum):** A special data type that represents a fixed set of named constants.

- **Purpose:** To define a collection of related constants (e.g., days of the week, cardinal directions, colors).
- **Type Safety:** Provides type safety compared to using plain `int` or `String` constants.
- **Enhancements:** Enums in Java are full-fledged classes. They can have:
  - Constructors
  - Methods (instance and static)
  - Fields

    - Implement interfaces
- **Declaration:** Uses the `enum` keyword.

**Example:**

```java
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

enum Status {
    PENDING("P"), PROCESSING("PR"), COMPLETED("C"), FAILED("F"); // Constants with
associated values

    private String code; // Field in enum

    private Status(String code) { // Constructor for enum constants
        this.code = code;
    }

    public String getCode() { // Method in enum
        return code;
    }

    public boolean isFinal() { // Another method
        return this == COMPLETED || this == FAILED;
    }
}

public class EnumDemo {
    public static void main(String[] args) {
        Day today = Day.MONDAY;
        System.out.println("Today is: " + today);

        switch (today) {
            case SATURDAY:
            case SUNDAY:
                System.out.println("It's a weekend.");
                break;
            default:
                System.out.println("It's a weekday.");
        }

        Status orderStatus = Status.PROCESSING;
        System.out.println("Order Status: " + orderStatus + ", Code: " +
orderStatus.getCode());
        System.out.println("Is order status final? " + orderStatus.isFinal());

        for (Status s : Status.values()) { // Iterate over all enum constants
            System.out.println(s.name() + " -> " + s.ordinal()); // name() and
ordinal() are built-in
        }
        }
    }
```

---

## Garbage collection in Java

**Garbage Collection (GC):** An automatic memory management process in Java. The JVM automatically reclaims memory occupied by objects that are no longer referenced by the program. This frees developers from manual memory deallocation, reducing memory leaks and dangling pointers.

- **Heap Memory:** GC primarily operates on the Heap area of the JVM memory, where objects are allocated.
- **"Garbage" Definition:** An object is considered "garbage" (eligible for garbage collection) if there are no longer any active references pointing to it from "roots" (e.g., local variables on the stack, static variables, JNI references).
- **Automatic Process:** GC runs automatically in the background as a low-priority daemon thread. Developers don't explicitly call `delete` or `free`.
- **Benefits:** Prevents memory leaks, simplifies development, enhances robustness.

### Requesting JVM to run garbage collection

While you cannot *force* GC to run immediately, you can *suggest* or *request* it to the JVM. The JVM decides if and when to honor the request.

- **`System.gc()`:** A static method that hints to the JVM that it would be a good time to run the garbage collector.

  ```
  System.gc(); // Suggests a garbage collection cycle
  ```

- **`Runtime.getRuntime().gc()`:** Another way to achieve the same hint.

  ```
  Runtime.getRuntime().gc(); // Also suggests a garbage collection cycle
  ```

**Important Note:** Do not rely on `System.gc()` or `Runtime.getRuntime().gc()` for critical resource management, as their execution is not guaranteed. For explicit resource cleanup (e.g., closing file streams, database connections), use `try-with-resources` or `finally` blocks to ensure resources are released reliably.

### Different ways to make object eligible for garbage collection

An object becomes eligible for garbage collection when it is no longer reachable from any active "root" references. Common ways to make an object eligible:

1. **Nulling a reference variable:**

- Setting a reference variable to `null` means it no longer points to any object. If this was the *only* reference to an object, that object becomes eligible.

```java
Object obj = new Object(); // Object created, obj refers to it
obj = null; // Object is now eligible for GC
```

2. **Re-assigning a reference variable:**
    - When a reference variable is assigned to a new object, its old object (if no other references exist) becomes eligible for GC.

```java
String s1 = new String("Hello"); // Object 1
String s2 = new String("World");  // Object 2

s1 = s2; // s1 now points to Object 2. Object 1 is eligible for GC (if no
other references to it).
```

3. **Island of Isolation (Circular Dependencies):**
    - A scenario where a group of objects reference each other, but are not referenced by any external "root" reference. This creates an "island" of unreachable objects.
    - Even though objects within the island still hold references to each other, the entire island is isolated from the active part of the program and becomes eligible for GC.

```java
class Link {
    Link next;
    Link(Link n) { this.next = n; }
}

public class IslandOfIsolation {
    public static void main(String[] args) {
        Link a = new Link(null);
        Link b = new Link(null);

        a.next = b; // a -> b
        b.next = a; // b -> a (circular reference)

        a = null; // Both objects 'a' and 'b' become unreachable from root
        b = null; // The "island" {a,b} is now eligible for GC
    }
}
```

### Finalize method

- **`finalize()` method:** A method inherited from the `java.lang.Object` class.
    - **Invocation:** It is called by the garbage collector *just before* an object is actually destroyed.

- **Purpose (Historical):** Intended for performing cleanup operations for non-Java resources (e.g., closing file handles, network connections) associated with an object, if these resources are not released by other means.
- **Problems and Deprecation:**
  - **Not Guaranteed:** `finalize()` is not guaranteed to be executed. If the JVM exits abruptly, or if the garbage collector doesn't run, `finalize()` might never be called.
  - **Unpredictable Timing:** Its execution time is unpredictable and can be delayed.
  - **Performance Overhead:** It can negatively impact GC performance.
  - **Rebirth of Objects:** A `finalize()` method can "resurrect" an object by making it reachable again, preventing its collection.
  - **Exceptions:** Exceptions thrown in `finalize()` are ignored.
- **Recommendation: Avoid using `finalize()`**. For resource cleanup, use `try-with-resources` statements for `AutoCloseable` resources, or implement explicit `close()` methods and call them in `finally` blocks.

---

## Methods, overloading, parameter passing, objects as parameters

### Methods

As previously discussed, methods define the behavior or actions that an object can perform.

- **Syntax:** `[access_specifier] [static/final/abstract] [return_type] methodName([parameters]) { // method body }`
- **Signature:** A method's signature consists of its name and the number, type, and order of its parameters. The return type is NOT part of the method signature.

### Overloading

- **Method Overloading:** A compile-time polymorphism feature where a class can have multiple methods with the same name, but different method signatures (different number of parameters, different data types of parameters, or different order of parameters).
- **Compile-time Resolution:** The compiler determines which overloaded method to call based on the arguments provided at the call site.
- **Return Type:** The return type *alone* is not sufficient to distinguish overloaded methods.
- **Example:** (from earlier section)

```java
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; } // Different parameter types
    int add(int a, int b, int c) { return a + b + c; } // Different number of parameters
}
```

### Parameter Passing

Java uses **pass-by-value** for all parameters. This means that a copy of the argument's value is passed to the method parameter.

- **Primitive Data Types:** When a primitive value is passed, a copy of the actual value is made. Any changes to the parameter inside the method do not affect the original variable.

  ```java
  void changePrimitive(int num) {
      num = 100; // Changes only the local copy
  }
  int x = 50;
  changePrimitive(x);
  System.out.println(x); // Output: 50
  ```

- **Reference Data Types (Objects):** When an object is passed, a copy of the *reference* (the memory address) to the object is made. Both the original reference variable and the method's parameter reference variable now point to the *same* object on the heap.
  - **Can modify object's state:** Changes made to the object's instance variables (state) through the method's parameter reference *will* affect the original object.
  - **Cannot re-point original reference:** The method cannot change which object the original reference variable points to. If you assign a new object to the method's parameter reference, it only changes that local copy.

  ```java
  class Point { int x, y; Point(int x, int y) { this.x = x; this.y = y; } }

  void changeObject(Point p) {
      p.x = 20; // Modifies the object pointed to by the original reference
      p = new Point(0, 0); // Re-points local 'p', does not affect original
  'myPoint'
  }
  Point myPoint = new Point(10, 10);
  changeObject(myPoint);
  System.out.println(myPoint.x); // Output: 20
  ```

### Memory management, garbage collection

This topic has been covered in detail in Session 9. Java handles memory management automatically through its Garbage Collector, which reclaims memory from objects that are no longer referenced. Developers do not manually allocate or deallocate memory.

### `this` keyword, static data and methods, block, scope, lifetime

- **`this` keyword:**
  - **Purpose:** A reference to the current object (the instance of the class on which the method is invoked or which is being constructed).
  - **Use Cases:**

        - **Disambiguation:** To distinguish between instance variables and local/parameter variables with the same name.
        - **Calling current class constructor:** `this(...)` is used for constructor chaining (must be the first statement in a constructor).
        - **Returning current class instance:** `return this;`
        - **Passing current instance as argument:** `someMethod(this);`

    ```java
    class Box {
        int width;
        Box(int width) {
            this.width = width; // 'this.width' refers to instance variable,
    'width' refers to parameter
        }
    }
    ```

- **Static data and methods:** Covered in Session 5. They belong to the class, not an object instance.
- **Block:** A group of zero or more statements enclosed in curly braces `{}`.
    - **Types:** Method body, loop body, `if`/`else` block, `try`/`catch`/`finally` block, static block, instance block.
- **Scope:** The region of a program where a variable is accessible.
    - **Class Scope:** Instance variables and static variables are accessible throughout the class.
    - **Method Scope:** Parameters and local variables are accessible only within the method.
    - **Block Scope:** Variables declared inside a block are accessible only within that block.
- **Lifetime:** The period during which a variable exists in memory.
    - **Local Variables:** Exist only while the method/block is executing.
    - **Instance Variables:** Exist as long as the object instance exists.
    - **Static Variables:** Exist from the time the class is loaded until the program terminates.

### JDK tools (Java Compiler, Java Runtime, Java Debugger, Javadoc)

The JDK (Java Development Kit) provides various command-line tools for developing and managing Java applications.

- `javac` **(Java Compiler):**
    - **Purpose:** Compiles Java source code (`.java` files) into Java bytecode (`.class` files).
    - **Usage:** `javac MyClass.java`
- `java` **(Java Runtime Launcher):**
    - **Purpose:** Executes compiled Java bytecode (`.class` files) by launching a JVM.
    - **Usage:** `java MyClass` (for a class with a `main` method).
- `jdb` **(Java Debugger):**
    - **Purpose:** A command-line debugger for Java programs. Allows you to set breakpoints, step through code, inspect variables, etc.
    - **Usage:** `jdb MyClass`
- `javadoc` **(API Documentation Generator):**
    - **Purpose:** Generates HTML documentation for Java source code comments (Javadoc comments).
    - **Usage:** `javadoc MyClass.java`

  - **Javadoc Comments:** Special multi-line comments starting with `/**` and ending with `*/`. Used to describe classes, methods, and fields. Support special tags like `@param`, `@return`, `@author`, `@see`.

---

## Packages

**Packages:** As discussed in Session 3, packages are Java's mechanism for organizing classes, interfaces, and subpackages into logical namespaces. They help prevent naming conflicts and improve code organization.

- **Declaration:** Uses the `package` keyword at the top of a Java source file.

```
package com.example.myproject;
// ... class definitions
```

- **Naming Convention:** Typically lowercase and follow reverse domain name notation (e.g., `com.mycompany.application.module`).
- `import` **Statement:** Used to access classes from other packages.
  - `import com.example.myproject.MyClass;` (imports a specific class)
  - `import com.example.myproject.*;` (imports all classes in a package - generally discouraged for clarity).
  - `java.lang` package is implicitly imported in all Java programs.

### Access Control Rules

Access control rules (using access modifiers) are applied at the package level, influencing visibility.

| Modifier | Class | Package (Default) | Subclass (different package) | World (different package, non-subclass) |
|---|---|---|---|---|
| `private` | Yes | No | No | No |
| `default` | Yes | Yes | No | No |
| `protected` | Yes | Yes | Yes | No |
| `public` | Yes | Yes | Yes | Yes |

- **Class-level Access:** A class itself can only be `public` or `default`. Only one `public` class per `.java` file, and its name must match the filename.

### Types of Inheritance (Recap from Session 6)

Java supports:

- **Single Inheritance:** `class B extends A {}`
- **Multilevel Inheritance:** `class C extends B {}` (where `B` extends `A`)
- **Hierarchical Inheritance:** `class B extends A {}`, `class C extends A {}`
- **Hybrid Inheritance:** Combinations of the above (where multiple inheritance of classes is avoided).

Java does **NOT** support multiple inheritance of *classes* directly. It supports multiple inheritance of *type* through interfaces.

### "IS-A" Relationship

- **Concept:** Inheritance establishes an "IS-A" relationship between a subclass and its superclass.
- **Meaning:** "A `Child` IS-A `Parent`." This means a `Child` object can be treated as a `Parent` object.
- **Polymorphism:** This relationship is fundamental to runtime polymorphism, where a superclass reference can point to a subclass object.

**Example:** If `Dog extends Animal`, then a `Dog` IS-A `Animal`.

```
Animal myAnimal = new Dog(); // Valid because Dog IS-A Animal
```

### Inheritance & Polymorphism

- **Inheritance** provides the mechanism for **Polymorphism**.
- When a subclass overrides a method from its superclass, and you have a superclass reference pointing to a subclass object, the specific method implementation that runs is determined by the *actual type of the object at runtime* (runtime polymorphism).

**Example:**

```
class Vehicle {
    void move() { System.out.println("Vehicle is moving."); }
}

class Car extends Vehicle {
    @Override
    void move() { System.out.println("Car is driving."); }
}

class Bicycle extends Vehicle {
    @Override
    void move() { System.out.println("Bicycle is pedaling."); }
}

public class InheritancePolymorphismDemo {
    public static void main(String[] args) {
        Vehicle[] vehicles = new Vehicle[3]; // Array of Vehicle references
        vehicles[0] = new Vehicle();
        vehicles[1] = new Car();     // Vehicle reference points to Car object
        vehicles[2] = new Bicycle(); // Vehicle reference points to Bicycle object

        for (Vehicle v : vehicles) {
            v.move(); // Runtime polymorphism in action:
                      // Calls Vehicle.move(), Car.move(), Bicycle.move()
respectively
        }
    }
}
```

---

## Compile Time Polymorphism (Recap from Session 6)

- **Method Overloading:** As discussed in Session 6 and 10, this is the primary way to achieve compile-time polymorphism. The compiler decides which method to invoke based on the method signature (name and parameter types/count) at compile time.

## Runtime Polymorphism (Dynamic method Dispatch) (Recap from Session 6)

- **Method Overriding:** The key mechanism. A subclass provides its own implementation for a method defined in its superclass.
- **Dynamic Method Dispatch:** The process by which a call to an overridden method is resolved at runtime, not at compile time. The JVM looks at the *actual type* of the object (not the type of the reference variable) to determine which version of the method to execute.
- **Requires:**
    1. Inheritance.
    2. Method Overriding (`@Override` annotation is recommended).
    3. A superclass reference variable holding a subclass object.

**Example:** (Repeated for emphasis)

```java
class Animal {
    void eat() { System.out.println("Animal eats food."); }
}

class Dog extends Animal {
    @Override
    void eat() { System.out.println("Dog eats kibble."); }
}

class Cat extends Animal {
    @Override
    void eat() { System.out.println("Cat eats fish."); }
}

public class DynamicMethodDispatchDemo {
    public static void main(String[] args) {
        Animal a = new Animal(); // Animal reference and object
        Animal d = new Dog();    // Animal reference but Dog object
        Animal c = new Cat();    // Animal reference but Cat object

        a.eat(); // Output: Animal eats food.
        d.eat(); // Output: Dog eats kibble. (Dynamic dispatch)
        c.eat(); // Output: Cat eats fish. (Dynamic dispatch)
    }
}
```

## Abstract Classes (Recap from Session 7)

- As discussed, abstract classes are partially implemented classes that cannot be instantiated directly. They serve as blueprints for subclasses and can contain both concrete and abstract methods.
- They force concrete subclasses to provide implementations for abstract methods.

## Final keyword (Recap from Session 8)

- The `final` keyword restricts modifications:
  - `final variable`: Becomes a constant.
  - `final method`: Cannot be overridden.
  - `final class`: Cannot be subclassed.

## Interfaces (Recap from Session 7 and 8)

- Interfaces define a contract and achieve 100% abstraction (for abstract methods).
- They support multiple inheritance of *type*.
- Since Java 8, they can have `default` and `static` methods.

## Dynamic method dispatch using Interfaces.

Runtime polymorphism (dynamic method dispatch) can also be achieved using interfaces. When an interface reference variable refers to an object of a class that implements that interface, the specific method implementation called is determined at runtime based on the actual object's type.

**Example:**

```java
interface Drawable {
    void draw();
}

class Circle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle.");
    }
}

class Square implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a Square.");
    }
}

public class InterfaceDynamicDispatchDemo {
    public static void main(String[] args) {
        Drawable d; // Declare an interface reference
        d = new Circle(); // d refers to a Circle object
        d.draw(); // Output: Drawing a Circle.

        d = new Square(); // d refers to a Square object
        d.draw(); // Output: Drawing a Square.

        // Can use an array of interface references
        Drawable[] shapes = new Drawable[2];
        shapes[0] = new Circle();
        shapes[1] = new Square();

        for (Drawable shape : shapes) {
            shape.draw(); // Dynamic dispatch on interface methods
        }
    }
}
```

---

## Inner classes

**Inner Classes:** A class defined within another class. They provide a way to logically group classes that are used only in one place, increase encapsulation, and make code more readable and maintainable.

- **Relationship:** An inner class is a member of its outer class, and it can access all members (including private ones) of the outer class.
- **Types of Inner Classes:**

1. **Nested Top-Level Class (Static Nested Class):**
    - **Declaration:** Declared with the `static` keyword inside an outer class.
    - **Access:** Can only access static members of the outer class directly. It cannot access non-static (instance) members of the outer class without an object of the outer class.
    - **Instantiation:** Can be instantiated without an object of the outer class: `OuterClass.StaticNestedClass nested = new OuterClass.StaticNestedClass();`
    - **Use Cases:** When a class is logically part of another class but does not require access to the outer class's instance members.
2. **Member Inner Class (Non-static Inner Class):**
    - **Declaration:** Declared without the `static` keyword inside an outer class.
    - **Access:** Can access all members (static and non-static, including private) of the outer class directly. It implicitly holds a reference to the outer class instance.
    - **Instantiation:** Requires an instance of the outer class to be instantiated: `OuterClass.InnerClass inner = outerObject.new InnerClass();`
    - **Use Cases:** When the inner class's existence is tightly coupled to an instance of the outer class, and it needs access to the outer instance's data.
3. **Local Inner Class:**
    - **Declaration:** Defined inside a method or any block.
    - **Scope:** Accessible only within the method/block where it is defined.
    - **Access:** Can access final or effectively final local variables of the enclosing block.
    - **Use Cases:** For very specialized classes that are only used within a specific method.

4. **Anonymous Inner Class:**
    - **Declaration:** A class without a name, defined and instantiated in a single expression. It typically implements an interface or extends a class.
    - **Purpose:** Used for one-time, simple implementations of interfaces or abstract classes, often for event handling.
    - **Syntax:** `new InterfaceOrClassName() { // implementation }`
    - **Use Cases:** Event listeners (e.g., `ActionListener`, `OnClickListener`), defining thread `Runnable`s, implementing functional interfaces with concise logic (often superseded by Lambda Expressions in Java 8+).

**Example:**

```
public class OuterClass {
    private int outerInstanceVar = 10;
    private static int outerStaticVar = 20;

    // 1. Static Nested Class
    static class StaticNestedClass {
        void display() {
            System.out.println("Static Nested Class: Outer static var = " +
outerStaticVar);
            // System.out.println(outerInstanceVar); // Compile-time error
        }
    }

    // 2. Member Inner Class
    class InnerClass {
        void display() {
            System.out.println("Inner Class: Outer instance var = " +
outerInstanceVar);
            System.out.println("Inner Class: Outer static var = " +
outerStaticVar);
        }
    }

    public void outerMethod() {
        // 3. Local Inner Class
        class LocalInnerClass {
            void display() {
                System.out.println("Local Inner Class: Outer instance var = " +
outerInstanceVar);
            }
        }
        LocalInnerClass local = new LocalInnerClass();
        local.display();
    }

    public static void main(String[] args) {
        // Static Nested Class usage
        OuterClass.StaticNestedClass staticNested = new
OuterClass.StaticNestedClass();
```

```
        staticNested.display();

        // Member Inner Class usage (requires outer instance)
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display();

        // Local Inner Class usage
        outer.outerMethod();

        // 4. Anonymous Inner Class (example with Runnable)
        Runnable myRunnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("Anonymous Inner Class: Running in a thread.");
                // Can access outerInstanceVar if 'outer' was effectively final
            }
        };
        new Thread(myRunnable).start();

        // Lambda expression (Java 8+) simplifies anonymous inner class for
functional interfaces
        Runnable lambdaRunnable = () -> System.out.println("Lambda: Running in a
thread.");
        new Thread(lambdaRunnable).start();
    }
}
```

### Exception Handling

**Exception Handling:** A mechanism in Java to manage runtime errors (exceptions) that disrupt the normal flow of a program. It allows programs to gracefully recover from errors, preventing crashes.

- **Error vs. Exception:**
  - **`Error`:** Represents serious problems that applications should not try to catch (e.g., `OutOfMemoryError`, `StackOverflowError`). Indicate severe issues with the JVM itself.
  - **`Exception`:** Represents conditions that a reasonable application might want to catch (e.g., `FileNotFoundException`, `NullPointerException`).

### Exception class inheritance hierarchy

All exceptions in Java are objects and are part of an inheritance hierarchy:

- **`Throwable`:** The superclass of all errors and exceptions.
  - **`Error`:** (Subclass of `Throwable`) For severe, unrecoverable problems.
  - **`Exception`:** (Subclass of `Throwable`) For recoverable problems.
    - **`RuntimeException` (Unchecked Exceptions):**
      - Subclasses of `RuntimeException` (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`).
      - The compiler does not force you to handle them (no `throws` clause needed, no `try-catch` required).

        - Usually indicate programming errors.
    - **Checked Exceptions:**
        - All `Exception` subclasses *except* `RuntimeException` and its subclasses (e.g., `IOException`, `SQLException`, `ClassNotFoundException`).
        - The compiler forces you to either handle them (`try-catch`) or declare that your method might throw them (`throws` clause).
        - Usually indicate external problems that are beyond the program's control but can be handled (e.g., file not found).

### `try`, `catch`, `finally`, `throw`, `throws`

These keywords form the core of Java's exception handling.

- **`try` block:**
    - Contains the code that might throw an exception.
    - If an exception occurs within the `try` block, control immediately transfers to the appropriate `catch` block.
- **`catch` block:**
    - Follows a `try` block.
    - Specifies the type of exception it can handle.
    - Contains code to execute when a specific exception is caught.
    - A `try` block can have multiple `catch` blocks for different exception types.
    - Order matters: more specific exceptions should be caught before more general ones.
- **`finally` block:**
    - Follows `try` and `catch` blocks.
    - Contains code that is *always* executed, regardless of whether an exception occurred in the `try` block or was caught by a `catch` block.
    - **Purpose:** Used for cleanup operations (e.g., closing file streams, database connections).
- **`throw` keyword:**
    - Used to explicitly `throw` an exception object from a method or block.
    - You create an `Exception` object (`new ExceptionType("message")`) and then `throw` it.
- **`throws` keyword:**
    - Used in a method signature to declare that a method *might throw* one or more checked exceptions.
    - It informs the caller that they need to either handle these exceptions or re-declare them.

**Example:**

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

public class ExceptionHandlingDemo {

    public static void divide(int numerator, int denominator) {
        try {
            int result = numerator / denominator; // May throw ArithmeticException (unchecked)
            System.out.println("Division result: " + result);
        } catch (ArithmeticException e) {
            System.err.println("Error: Cannot divide by zero. " + e.getMessage());
        } finally {
            System.out.println("Division operation attempted.");
        }
    }

    public static void readFile(String fileName) throws IOException { // Declares
it might throw IOException
        FileReader reader = null;
        try {
            reader = new FileReader(fileName); // May throw FileNotFoundException
(checked - subclass of IOException)
            int charRead;
            while ((charRead = reader.read()) != -1) {
                System.out.print((char) charRead);
            }
            System.out.println("\nFile read successfully.");
        } catch (IOException e) { // Catches IOException (includes
FileNotFoundException)
            System.err.println("Error reading file: " + e.getMessage());
            throw e; // Re-throws the caught exception (or throw a new custom
exception)
        } finally {
            if (reader != null) {
                try {
                    reader.close(); // Ensure reader is closed
                } catch (IOException closeEx) {
                    System.err.println("Error closing reader: " +
closeEx.getMessage());
                }
            }
            System.out.println("File operation finished.");
        }
    }

    public static void main(String[] args) {
        divide(10, 2);
        divide(10, 0); // Triggers ArithmeticException

        try {
            readFile("non_existent_file.txt"); // This call must be in a try-catch
or main must declare throws IOException
        } catch (IOException e) {
            System.err.println("Caught IOException in main: " + e.getMessage());
        }
    }
}
```

Custom exception classes

You can create your own custom exception classes by extending `Exception` (for checked exceptions) or `RuntimeException` (for unchecked exceptions).

- **Purpose:** To provide more specific and meaningful error information tailored to your application's domain.
- **Naming Convention:** End with `Exception`.

**Example:**

```java
// Custom Checked Exception
class InsufficientFundsException extends Exception {
    private double requiredAmount;

    public InsufficientFundsException(String message, double requiredAmount) {
        super(message);
        this.requiredAmount = requiredAmount;
    }

    public double getRequiredAmount() {
        return requiredAmount;
    }
}

// Custom Unchecked Exception
class InvalidInputException extends RuntimeException {
    public InvalidInputException(String message) {
        super(message);
    }
}

class BankAccount {
    private double balance;

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

    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount <= 0) {
            throw new InvalidInputException("Withdrawal amount must be positive."); // Unchecked, no need to declare throws
        }
        if (balance < amount) {
            throw new InsufficientFundsException("Not enough balance.", amount - balance); // Checked, must declare throws
        }
        balance -= amount;
        System.out.println("Withdrew " + amount + ". New balance: " + balance);
    }
}

public class CustomExceptionDemo {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);

        try {
            account.withdraw(200);
            account.withdraw(1200); // This will throw InsufficientFundsException
            account.withdraw(-50); // This will throw InvalidInputException
(runtime)
        } catch (InsufficientFundsException e) {
            System.err.println("Caught InsufficientFundsException: " +
e.getMessage() + " (Needed: " + e.getRequiredAmount() + ")");
        } catch (InvalidInputException e) { // Catch unchecked exception if
desired, though not required
            System.err.println("Caught InvalidInputException: " + e.getMessage());
        } catch (Exception e) { // Generic catch for any other unexpected
exceptions
            System.err.println("An unexpected error occurred: " + e.getMessage());
        }
    }
}
```

---

## The `java.lang` package, Object, Number, Math, System

The `java.lang` package is automatically imported into every Java program. It provides fundamental classes that are essential to the design of the Java language.

- `Object` **class:**
  - **Root of Hierarchy:** The topmost class in the Java class hierarchy. Every class in Java directly or indirectly inherits from `Object`.
  - **Common Methods (inherited by all classes):**
    - `equals(Object obj)`: Compares if two objects are logically equal. Default implementation compares references (`==`). Should be overridden for custom equality.
    - `hashCode()`: Returns a hash code value for the object. Must be consistent with `equals()`.
    - `toString()`: Returns a string representation of the object. Default is `ClassName@hashCode`. Should be overridden for meaningful representation.
    - `getClass()`: Returns the runtime class of the object.
    - `notify()`, `notifyAll()`, `wait()`: For inter-thread communication.
    - `clone()`: Creates a shallow copy of the object (requires `Cloneable` interface).
    - `finalize()`: (Deprecated) Called by GC before destruction.
- `Number` **class:**
  - **Abstract Class:** The abstract superclass of all wrapper classes for numeric primitive types (`Integer`, `Double`, `Float`, `Long`, `Short`, `Byte`).
  - **Purpose:** Provides a common interface for converting between different numeric primitive types (e.g., `intValue()`, `doubleValue()`).
- `Math` **class:**
  - **Purpose:** Provides static methods for performing common mathematical operations.
  - **Characteristics:** All methods are `static` (no need to create `Math` object) and `final`.

- **Examples:** `Math.sqrt()`, `Math.pow()`, `Math.abs()`, `Math.min()`, `Math.max()`, `Math.random()`, `Math.round()`, `Math.PI`, `Math.E`.
- `System` **class:**
  - **Purpose:** Provides static methods and fields for standard system-related operations.
  - **Characteristics:** All methods/fields are `static` and the class is `final`.
  - **Examples:**
    - `System.out`: Standard output stream (`PrintStream` object, used for `System.out.println()`).
    - `System.err`: Standard error stream.
    - `System.in`: Standard input stream.
    - `System.exit(int status)`: Terminates the JVM.
    - `System.currentTimeMillis()`: Returns current time in milliseconds.
    - `System.gc()`: Suggests garbage collection.
    - `System.getProperty(String key)`: Gets system properties.

### Wrapper classes

Wrapper classes provide object representations for primitive data types.

- **Purpose:**
  - To use primitive values in contexts where objects are required (e.g., storing primitives in Collections Framework classes like `ArrayList`, `HashMap`).
  - To provide utility methods for primitive values (e.g., parsing strings to numbers, converting to binary/hex).
- **Classes:**
  - `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`.
- **Autoboxing and Unboxing (Java 5+):**
  - **Autoboxing:** Automatic conversion of a primitive value into its corresponding wrapper object.
  - **Unboxing:** Automatic conversion of a wrapper object into its corresponding primitive value.

**Example:**

```java
// Autoboxing
Integer intObject = 10; // int 10 is autoboxed to an Integer object
Double doubleObject = 3.14; // double 3.14 is autoboxed to a Double object

// Unboxing
int primitiveInt = intObject; // Integer object is unboxed to int
double primitiveDouble = doubleObject; // Double object is unboxed to double

// Using wrapper class methods
String numStr = "123";
int parsedInt = Integer.parseInt(numStr); // Convert string to int
String binaryStr = Integer.toBinaryString(10); // Convert int to binary string
```

### String class, StringBuffer & StringBuilder class String pool

- `String` **class:**

    - **Purpose:** Represents immutable sequences of characters.
    - **Immutability:** Once a `String` object is created, its content cannot be changed. Any operation that appears to modify a `String` actually creates a new `String` object.
    - **Memory:** `String` literals are stored in the String Pool (a special area in the heap/PermGen/Metaspace), which helps in memory optimization by reusing identical string literals.
    - **Use Cases:** For fixed, unchangeable text.

    ```
    String s1 = "Hello"; // "Hello" is created in String Pool
    String s2 = "Hello"; // s2 refers to the same "Hello" from String Pool
    String s3 = new String("World"); // "World" is created on heap, outside pool
    (unless optimized by JVM)
    s1 = s1 + " World"; // Creates a NEW String object "Hello World", s1 now
    points to it.
    ```

- `StringBuffer` **class:**
    - **Purpose:** Represents mutable sequences of characters. Thread-safe (synchronized).
    - **Mutability:** Its content can be modified without creating new objects.
    - **Thread-Safety:** Methods are synchronized, making it safe for use in multithreaded environments. This comes with a performance overhead.
    - **Use Cases:** When string manipulation is needed in a multithreaded context.
- `StringBuilder` **class:**
    - **Purpose:** Represents mutable sequences of characters. Not thread-safe (non-synchronized).
    - **Mutability:** Same as `StringBuffer`.
    - **Performance:** Faster than `StringBuffer` because its methods are not synchronized.
    - **Use Cases:** When string manipulation is needed in a single-threaded context (most common).

**Example:**

```
// String (immutable)
String str = "Java";
str.concat(" Programming"); // A new String "Java Programming" is created, but str
still points to "Java"
System.out.println(str); // Output: Java

// StringBuffer (mutable, thread-safe)
StringBuffer sb = new StringBuffer("Java");
sb.append(" Programming"); // Modifies the original StringBuffer object
System.out.println(sb); // Output: Java Programming

// StringBuilder (mutable, not thread-safe)
StringBuilder sbd = new StringBuilder("Java");
sbd.append(" Programming"); // Modifies the original StringBuilder object
System.out.println(sbd); // Output: Java Programming
```

## The `java.util` Package

The `java.util` package provides utility classes for various purposes, including data structures, date/time, random numbers, and more.

### `ArrayList`, `LinkedList`, `Vector`, `HashSet`, `LinkedHashSet`, `TreeSet`, `HashMap`, `TreeMap`

These are key classes from the Java Collections Framework, providing powerful ways to store and manipulate groups of objects.

#### `List` Interface (Ordered, Allows Duplicates)

- **`ArrayList`:**
  - **Underlying Data Structure:** Dynamic array (resizable array).
  - **Access:** Fast random access by index (`get(index)`).
  - **Insertion/Deletion:** Slow for insertions/deletions in the middle (requires shifting elements). Fast at the end.
  - **Thread-Safety:** Not synchronized (not thread-safe).
  - **Use Cases:** Frequent random access, adding/removing elements mostly at the end.
- **`LinkedList`:**
  - **Underlying Data Structure:** Doubly linked list.
  - **Access:** Slow random access by index (requires traversal from beginning/end).
  - **Insertion/Deletion:** Fast for insertions/deletions anywhere in the list.
  - **Thread-Safety:** Not synchronized.
  - **Use Cases:** Frequent insertions/deletions in the middle, implementing queues/stacks.
- **`Vector`:**
  - **Underlying Data Structure:** Dynamic array.
  - **Legacy:** Older class, effectively superseded by `ArrayList`.
  - **Thread-Safety:** Synchronized (thread-safe). This makes it slower than `ArrayList` in single-threaded environments.
  - **Use Cases:** When thread-safety is absolutely required for a resizable array (though `Collections.synchronizedList(new ArrayList<>())` is preferred).

#### `Set` Interface (Unordered, No Duplicates)

- **`HashSet`:**
  - **Underlying Data Structure:** Hash table (backed by `HashMap`).
  - **Order:** Does not maintain insertion order.
  - **Duplicates:** Does not allow duplicate elements.
  - **Performance:** Fast for add, remove, contains (O(1) on average). Requires objects to correctly implement `hashCode()` and `equals()`.
  - **Use Cases:** Storing unique elements where order doesn't matter, checking for existence quickly.
- **`LinkedHashSet`:**
  - **Underlying Data Structure:** Hash table with a linked list.
  - **Order:** Maintains insertion order.
  - **Duplicates:** Does not allow duplicate elements.
  - **Performance:** Slightly slower than `HashSet` due to linked list overhead, but still efficient.
  - **Use Cases:** Storing unique elements while preserving insertion order.
- **`TreeSet`:**

    - **Underlying Data Structure:** Red-black tree.
    - **Order:** Stores elements in sorted (natural or custom) order.
    - **Duplicates:** Does not allow duplicate elements.
    - **Performance:** Slower than `HashSet` (O(log n) for operations). Requires elements to implement `Comparable` or provide a `Comparator`.
    - **Use Cases:** Storing unique elements in sorted order, performing range queries.

#### `Map` Interface (Key-Value Pairs)

- `HashMap`:
    - **Underlying Data Structure:** Hash table.
    - **Order:** Does not maintain insertion order.
    - **Keys/Values:** Unique keys, values can be duplicated. Keys must be unique and implement `hashCode()` and `equals()`.
    - **Performance:** Fast for put, get, remove (O(1) on average).
    - **Use Cases:** Storing key-value pairs for fast lookup, caching.
- `TreeMap`:
    - **Underlying Data Structure:** Red-black tree.
    - **Order:** Stores entries in sorted (natural or custom) order based on keys.
    - **Performance:** Slower than `HashMap` (O(log n) for operations). Requires keys to implement `Comparable` or provide a `Comparator`.
    - **Use Cases:** Storing key-value pairs in sorted order of keys, performing range queries on keys.

---

## Introduction to Generics

**Generics:** A feature introduced in Java 5 that allows types (classes, interfaces, and methods) to operate on objects of various types while providing compile-time type safety. They enable you to write flexible, reusable code.

- **Problem Solved (before Generics):** Before generics, collections (like `ArrayList`) stored `Object`s. This meant:

    - No type safety at compile time (you could add any object type).
    - Required explicit casting when retrieving elements, leading to `ClassCastException` at runtime.

    ```
    // Pre-Generics
    ArrayList list = new ArrayList();
    list.add("Hello");
    list.add(123); // No compile-time error
    String s = (String) list.get(1); // Runtime ClassCastException
    ```

- **Benefits of Generics:**

    - **Compile-time Type Safety:** Catches type-mismatch errors at compile time, preventing `ClassCastException` at runtime.

 - **Eliminates Casts:** No need for explicit type casting when retrieving elements from generic collections.
 - **Code Reusability:** Write generic algorithms that work on various types.

### Generic classes & interfaces.

You can define classes and interfaces with type parameters.

- **Syntax:** Use angle brackets (`< >`) to declare type parameters (e.g., `<T>`, `<E>`, `<K, V>`). By convention, single uppercase letters are used.

**Example (Generic Class):**

```java
// Generic Box class that can hold any type of object
public class Box<T> { // T is a type parameter
    private T content;

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }

    public void setContent(T content) {
        this.content = content;
    }
}

public class GenericClassDemo {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>(10); // Box holds Integer
        System.out.println(integerBox.getContent()); // No cast needed
        // integerBox.setContent("Hello"); // Compile-time error: type safety

        Box<String> stringBox = new Box<>("Java Generics"); // Box holds String
        System.out.println(stringBox.getContent());
    }
}
```

**Example (Generic Interface):**

```java
interface MyList<E> { // E is an element type parameter
    void add(E element);
    E get(int index);
    int size();
}

// Concrete implementation
```

```java
class MyArrayList<E> implements MyList<E> {
    private Object[] elements = new Object[10];
    private int size = 0;

    @Override
    public void add(E element) {
        elements[size++] = element;
    }

    @Override
    @SuppressWarnings("unchecked") // Suppress warning for unsafe cast from Object
to E
    public E get(int index) {
        return (E) elements[index];
    }

    @Override
    public int size() {
        return size;
    }
}
```

### Wildcard syntax

Wildcards (`?`) are used in generic code to relax the restrictions on a type parameter.

- **`? extends T` (Upper Bounded Wildcard):**
    - **Meaning:** Represents an unknown type that is `T` or a *subclass* of `T`.
    - **Usage:** Can *read* (get) elements of type `T` or its supertypes from the collection. You **cannot add** (put) elements into such a collection (except for `null`). This follows the **PECS (Producer-Extends, Consumer-Super)** principle.
    - **Use Cases:** For methods that consume `T` objects (e.g., a method that sums up numbers, where the list can contain `Integer`s, `Double`s, etc., all extending `Number`).

    ```java
    public void printNumbers(List<? extends Number> list) {
        for (Number n : list) {
            System.out.println(n);
        }
        // list.add(new Integer(5)); // Compile-time error
    }
    ```

- **`? super T` (Lower Bounded Wildcard):**
    - **Meaning:** Represents an unknown type that is `T` or a *superclass* of `T`.
    - **Usage:** Can *add* (put) elements of type `T` (or a subclass of `T`) into the collection. You **cannot get** (read) elements from such a collection without casting to `Object`.
    - **Use Cases:** For methods that produce `T` objects (e.g., a method that adds `Integer`s to a list, where the list can accept `Number`s or `Object`s).

```
public void addIntegers(List<? super Integer> list) {
    list.add(10);
    list.add(20);
    // Integer num = list.get(0); // Compile-time error: type is ? super
Integer
    Object obj = list.get(0); // Can only retrieve as Object
}
```

- `?` **(Unbounded Wildcard):**
    - **Meaning:** Represents an unknown type. Similar to `List<? extends Object>`.
    - **Usage:** Can only read `Object`s from the collection. You cannot add elements.
    - **Use Cases:** When the method logic doesn't depend on the actual type parameter (e.g., `printList(List<?> list)`).

### Inheritance in Generics

- **Rule:** If `TypeB` is a subtype of `TypeA`, then `List<TypeB>` is **NOT** a subtype of `List<TypeA>`. Generics are invariant by default.

```
List<Integer> intList = new ArrayList<>();
// List<Number> numList = intList; // Compile-time error: List<Integer> is
NOT a List<Number>
```

- This is crucial for type safety. If `List<Integer>` were a `List<Number>`, you could add a `Double` to `numList`, which would then be present in `intList`, leading to a `ClassCastException` when an `Integer` was expected.
- Wildcards help introduce covariance (`? extends`) and contravariance (`? super`).

### The Java 2 Collection Framework

The Java Collections Framework (JCF) is a set of interfaces and classes in the `java.util` package that provides a unified architecture for representing and manipulating collections of objects.

- **Core Interfaces:**
    - **`Collection`:** The root interface for all collections. Defines common operations like `add()`, `remove()`, `contains()`, `size()`.
    - **`List`:** (Extends `Collection`) Ordered collection (maintains insertion order), allows duplicate elements. Implemented by `ArrayList`, `LinkedList`, `Vector`.
    - **`Set`:** (Extends `Collection`) Unordered collection, does not allow duplicate elements. Implemented by `HashSet`, `LinkedHashSet`, `TreeSet`.
    - **`Queue`:** (Extends `Collection`) Designed for holding elements prior to processing (FIFO - First-In, First-Out). Implemented by `LinkedList`, `PriorityQueue`.
    - **`Map`:** (NOT `Collection` but part of JCF) Stores key-value pairs. Keys must be unique. Implemented by `HashMap`, `TreeMap`, `LinkedHashMap`, `Hashtable`.

### List, Set & Map Functionality (Recap from Session 15 & 16)

- **List Functionality:**
  - Indexed access (`get(int)`, `set(int, E)`).
  - Iteration in order.
  - Add/remove at specific positions.
- **Set Functionality:**
  - Add/remove elements.
  - Check for existence (`contains()`).
  - Mathematical set operations (union, intersection, difference) using methods like `addAll()`, `retainAll()`, `removeAll()`.
- **Map Functionality:**
  - Store key-value pairs (`put(K, V)`).
  - Retrieve value by key (`get(K)`).
  - Remove entry by key (`remove(K)`).
  - Check for key/value existence (`containsKey()`, `containsValue()`).
  - Get sets of keys, values, or entries.

### Sorting using Natural Ordering & Custom Ordering

The Collections Framework provides powerful ways to sort elements.

- **Natural Ordering (`Comparable` interface):**
  - **Purpose:** For objects that have a natural, inherent order (e.g., numbers, strings, dates).
  - **Implementation:** The class whose objects are to be sorted must implement the `Comparable<T>` interface and override its `compareTo(T other)` method.
    - Returns a negative integer, zero, or a positive integer if `this` object is less than, equal to, or greater than `other`.
  - **Usage:**
    - `Collections.sort(List<T> list)`: Sorts a list of `Comparable` objects.
    - `Arrays.sort(T[] array)`: Sorts an array of `Comparable` objects.
    - `TreeSet`, `TreeMap`: Automatically sort elements/keys based on natural ordering.

**Example (Natural Ordering):**

```java
class Person implements Comparable<Person> {
    String name;
    int age;

    public Person(String name, int age) { this.name = name; this.age = age; }

    // Natural order: Sort by age, then by name if ages are same
    @Override
    public int compareTo(Person other) {
        int ageComparison = Integer.compare(this.age, other.age);
        if (ageComparison != 0) {
            return ageComparison;
        }
        return this.name.compareTo(other.name);
    }
    @Override
    public String toString() { return "Person{" + "name='" + name + '\'' + ",
age=" + age + '}'; }
}

public class NaturalOrderingDemo {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 30));

        Collections.sort(people); // Sorts using Person's compareTo method
        for (Person p : people) { System.out.println(p); }
        // Output: Bob (25), Alice (30), Charlie (30) (assuming Charlie comes
after Alice alphabetically)
    }
}
```

- **Custom Ordering (`Comparator` interface):**
  - **Purpose:** For objects that do not have a natural order, or when you need different sorting criteria.
  - **Implementation:** Create a separate class (often an anonymous inner class or a lambda expression) that implements the `Comparator<T>` interface and overrides its `compare(T o1, T o2)` method.
  - **Usage:**
    - `Collections.sort(List<T> list, Comparator<T> c)`
    - `Arrays.sort(T[] array, Comparator<T> c)`
    - Pass a `Comparator` to `TreeSet` or `TreeMap` constructors.
  - **Java 8+ Enhancements:** `Comparator.comparing()`, `thenComparing()`, `reversed()`.

**Example (Custom Ordering):**

```
// Person class without Comparable
// class Person { ... }

public class CustomOrderingDemo {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 30));

        // Sort by name descending using a Comparator (Lambda Expression)
        Collections.sort(people, (p1, p2) -> p2.name.compareTo(p1.name));
        for (Person p : people) { System.out.println(p); }
        // Output: Charlie (30), Bob (25), Alice (30)

        // Sort by age ascending, then by name ascending (using
Comparator.comparing)
people.sort(Comparator.comparing(Person::getAge).thenComparing(Person::getName));
        for (Person p : people) { System.out.println(p); }
        // Output: Bob (25), Alice (30), Charlie (30)
    }
}
```

---

## The `java.io` Package

The `java.io` package provides classes for system input and output operations, dealing with data streams.

### Byte Streams and Unicode Character Streams

Java I/O is based on streams, which represent a sequence of data.

- **Byte Streams:**
  - **Purpose:** Read/write raw bytes (8-bit bytes). Suitable for binary data (images, audio, video) or when character encoding is not a concern.
  - **Classes:**
    - **Input:** `InputStream` (abstract base class), `FileInputStream`, `ByteArrayInputStream`, `BufferedInputStream`, `ObjectInputStream`.
    - **Output:** `OutputStream` (abstract base class), `FileOutputStream`, `ByteArrayOutputStream`, `BufferedOutputStream`, `ObjectOutputStream`.
- **Unicode Character Streams:**
  - **Purpose:** Read/write characters (16-bit Unicode characters). Suitable for text data, automatically handling character encoding conversions.
  - **Classes:**
    - **Input:** `Reader` (abstract base class), `FileReader`, `BufferedReader`, `InputStreamReader`.
    - **Output:** `Writer` (abstract base class), `FileWriter`, `PrintWriter`, `OutputStreamWriter`, `BufferedWriter`.
  - **Bridge Streams:** `InputStreamReader` and `OutputStreamWriter` act as bridges between byte streams and character streams, allowing you to specify character encodings.

### File Handling

The `File` class in `java.io` represents a file or directory path. It does not provide methods for reading or writing data but for manipulating file/directory properties.

- **`File` Class:**
  - **Purpose:** To represent file and directory pathnames. Provides methods for:
    - Creating/deleting files/directories (`createNewFile()`, `delete()`, `mkdir()`, `mkdirs()`).
    - Checking existence, readability, writability (`exists()`, `canRead()`, `canWrite()`).
    - Getting file info (`getName()`, `getPath()`, `length()`, `isDirectory()`, `isFile()`).
    - Listing directory contents (`listFiles()`).

**Example:**

```java
import java.io.File;

public class FileHandlingDemo {
    public static void main(String[] args) {
        File file = new File("example.txt");

        try {
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }

            System.out.println("Absolute path: " + file.getAbsolutePath());
            System.out.println("File size: " + file.length() + " bytes");

            // Deleting the file
            // if (file.delete()) {
            //     System.out.println("File deleted successfully.");
            // } else {
            //     System.out.println("Failed to delete the file.");
            // }

        } catch (IOException e) {
            System.err.println("An error occurred: " + e.getMessage());
        }

        File directory = new File("my_directory");
        if (directory.mkdir()) {
            System.out.println("Directory created: " + directory.getName());
        } else {
            System.out.println("Directory already exists or failed to create.");
        }
    }
}
```

### `BufferedReader`, `PrintWriter` streams

These are common character streams that provide buffering and convenience methods for reading/writing text.

- **`BufferedReader`:**
    - **Purpose:** Reads text from a character-input stream, buffering characters to provide for the efficient reading of characters, arrays, and lines.
    - **Common Use:** Wrapping `FileReader` or `InputStreamReader`.
    - **Methods:** `readLine()` (reads a line of text), `read()`, `close()`.
    - **Example (Reading a file line by line):**

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

public class BufferedReaderDemo {
    public static void main(String[] args) {
        String fileName = "test.txt";
        // Create a dummy file for testing
        try (PrintWriter writer = new PrintWriter(fileName)) {
            writer.println("Line 1");
            writer.println("Line 2");
        } catch (IOException e) { e.printStackTrace(); }

        try (BufferedReader reader = new BufferedReader(new
FileReader(fileName))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println("Read: " + line);
            }
        } catch (IOException e) {
            System.err.println("Error reading file: " +
e.getMessage());
        }
    }
}
```

- `PrintWriter`:
  - **Purpose:** Writes formatted representations of objects to a text-output stream. Provides convenience methods for printing various types (strings, numbers, etc.) and `println()` for new lines.
  - **Common Use:** Wrapping `FileWriter` or `OutputStreamWriter`.
  - **Methods:** `print()`, `println()`, `printf()`, `flush()`, `close()`.
  - **Example (Writing formatted text to a file):**

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

public class PrintWriterDemo {
    public static void main(String[] args) {
        try (PrintWriter writer = new PrintWriter(new
FileWriter("output.txt"))) {
            writer.println("Hello, Java I/O!");
            writer.printf("The value of PI is %.2f%n", Math.PI);
            writer.print("This is on the same line.");
        } catch (IOException e) {
            System.err.println("Error writing to file: " +
e.getMessage());
            }
        }
    }
```

- `try-with-resources`: (Java 7+) The `try (resource_declaration)` block automatically closes resources that implement `AutoCloseable` (like `BufferedReader`, `PrintWriter`, `FileReader`, `FileWriter`) when the block exits, whether normally or due to an exception. **Highly recommended for resource management.**

### Persistence of objects

Persistence refers to the ability of an object to outlive the process that created it. In Java, this means saving the state of an object so it can be restored later.

- **Mechanisms:**
  - **Serialization:** Converting an object's state into a byte stream.
  - **Database Storage:** Storing object data in a relational or NoSQL database (often using ORM tools).
  - **File Storage:** Saving object data to a file (could be custom format or JSON/XML).

### Object Serialization & deserialization

**Object Serialization:** The process of converting an object's state (its fields) into a sequence of bytes. This byte sequence can then be stored in a file, sent over a network, or stored in a database.

- `Serializable` **Interface:** A marker interface (`java.io.Serializable`). A class must implement this interface to be serializable. If a class implements `Serializable`, all its non-`transient` instance fields are serialized.
- `ObjectOutputStream`: Used to write serialized objects to an `OutputStream`.
- `transient` **keyword:** Marks a field to be excluded from serialization.
- `static` **fields:** Are not serialized (they belong to the class, not the object state).

**Object Deserialization:** The reverse process: reconstructing an object from its serialized byte stream.

- `ObjectInputStream`: Used to read serialized objects from an `InputStream`.
- `readObject()` **method:** Reads an object from the stream. Returns an `Object`, which needs to be cast back to the original type.

**Example:**

```
import java.io.*;

// 1. Create a class that implements Serializable
class Employee implements Serializable {
    private static final long serialVersionUID = 1L; // Recommended for versioning
    int id;
    String name;
    transient double salary; // 'salary' will NOT be serialized
```

```java
    static String company = "ABC Corp"; // Static fields are not serialized

    public Employee(int id, String name, double salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + ", company=" + company + "]";
    }
}

public class ObjectSerializationDemo {
    public static void main(String[] args) {
        Employee emp = new Employee(101, "Alice", 75000.0);
        System.out.println("Original object: " + emp);

        // Serialization
        try (FileOutputStream fileOut = new FileOutputStream("employee.ser");
             ObjectOutputStream out = new ObjectOutputStream(fileOut)) {
            out.writeObject(emp); // Write the object to the stream
            System.out.println("Employee object serialized to employee.ser");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialization
        Employee deserializedEmp = null;
        try (FileInputStream fileIn = new FileInputStream("employee.ser");
             ObjectInputStream in = new ObjectInputStream(fileIn)) {
            deserializedEmp = (Employee) in.readObject(); // Read the object from the stream
            System.out.println("Employee object deserialized.");
            System.out.println("Deserialized object: " + deserializedEmp);
            // Note: salary will be 0.0 (default for double), company will be "ABC Corp" (static, not from serialization)
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

---

## Multithreaded programming in Java

**Multithreading:** The ability of a program to execute multiple parts (threads) concurrently. This allows a program to perform multiple tasks seemingly simultaneously, improving responsiveness and efficiency, especially for long-running or I/O-bound operations.

- **Thread:** The smallest unit of execution that can be managed by a scheduler. A single process can have multiple threads.
- **Benefits:**
    - **Responsiveness:** UI remains responsive while background tasks run.
    - **Resource Utilization:** Better utilization of CPU cores (parallelism).
    - **Responsiveness to I/O:** Can perform other tasks while waiting for I/O operations.

### Multitasking - Process Based Vs Thread Based

- **Process-Based Multitasking:**
    - **Concept:** The operating system runs multiple independent programs (processes) concurrently (e.g., running a browser, a word processor, and a music player at the same time).
    - **Memory:** Each process has its own separate memory space. IPC (Inter-Process Communication) is required to share data.
    - **Overhead:** High overhead for creation, switching, and communication.
- **Thread-Based Multitasking (Multithreading):**
    - **Concept:** A single program (process) runs multiple threads concurrently within its own memory space.
    - **Memory:** All threads within the same process share the same memory space (heap, method area). Each thread has its own stack and PC register.
    - **Overhead:** Lower overhead for creation, switching, and communication compared to processes.
    - **Challenge:** Requires careful synchronization to prevent data corruption when multiple threads access shared resources.

### Thread state transitions

A thread can exist in several states during its lifetime:

1. **New:** The thread has been created (`new Thread()`) but has not yet started execution.
2. **Runnable:** The thread is ready to run and is waiting for the CPU to be allocated. It might be currently executing or waiting for its turn.
    - Entered by `start()`.
    - Can move to `Running` (gets CPU) or `Waiting/Blocked`.
3. **Running:** The thread is currently executing on the CPU.
4. **Blocked (Waiting/Timed Waiting):** The thread is temporarily inactive and not eligible to run. It's waiting for some event to occur.
    - **Blocked:** Waiting for a monitor lock (e.g., trying to enter a `synchronized` block that is held by another thread).
    - **Waiting:** Waiting indefinitely for another thread to perform a particular action (e.g., `Object.wait()`, `Thread.join()`).
    - **Timed Waiting:** Waiting for a specified time interval or until another thread performs an action (e.g., `Thread.sleep()`, `Object.wait(long)`, `Thread.join(long)`).
5. **Terminated (Dead):** The thread has completed its execution or has been abnormally terminated. It cannot be restarted.

### The `Thread` class & its API

- **`Thread` class:** (`java.lang.Thread`) The primary class for creating and controlling threads in Java.

- **Creating Threads (by extending `Thread`):**
  1. Create a class that extends `Thread`.
  2. Override the `run()` method with the code the thread will execute.
  3. Create an instance of your class.
  4. Call the `start()` method on the instance (this calls `run()` on a new thread).

**Example:**

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

public class ThreadClassDemo {
    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // New state
        t1.start(); // Moves to Runnable state, then Running
        for (int i = 0; i < 5; i++) {
            System.out.println("Main Thread: " + i);
            try { Thread.sleep(500); } catch (InterruptedException e) {
e.printStackTrace(); }
        }
        System.out.println("Main Thread finished.");
    }
}
```

- **Common `Thread` API methods:**
  - `start()`: Starts the execution of the thread.
  - `run()`: Contains the code executed by the thread.
  - `sleep(long milliseconds)`: Pauses the current thread for a specified time.
  - `join()`: Waits for a thread to die.
  - `interrupt()`: Interrupts a thread.
  - `isAlive()`: Checks if a thread is alive.
  - `setPriority()`, `getPriority()`: Sets/gets thread priority.
  - `getName()`, `setName()`: Gets/sets thread name.
  - `currentThread()`: Returns a reference to the currently executing thread object.

## The `Runnable` interface

- **`Runnable` interface:** (`java.lang.Runnable`) A functional interface with a single abstract method `run()`. Provides a way to define a task that can be executed by a thread.
- **Creating Threads (by implementing `Runnable`):**

   1. Create a class that implements `Runnable`.
   2. Override the `run()` method.
   3. Create an instance of your `Runnable` class.
   4. Pass this `Runnable` instance to a `Thread` constructor.
   5. Call `start()` on the `Thread` object.
- **Advantages of `Runnable` over extending `Thread`:**
   - **Flexibility:** Your class can still extend another class (Java only supports single inheritance).
   - **Separation of Concerns:** Separates the task (what to run) from the thread (how to run it).
   - **Resource Sharing:** Easier to share common objects among multiple threads by passing them to the `Runnable`'s constructor.

**Example:**

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Runnable Thread: " + i);
            try { Thread.sleep(600); } catch (InterruptedException e) {
e.printStackTrace(); }
        }
        System.out.println("Runnable Thread finished.");
    }
}

public class RunnableDemo {
    public static void main(String[] args) {
        Runnable myTask = new MyRunnable();
        Thread t2 = new Thread(myTask); // Pass Runnable to Thread constructor
        t2.start();

        // Using Lambda for Runnable (Java 8+)
        Thread t3 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Lambda Thread: " + i);
                try { Thread.sleep(700); } catch (InterruptedException e) {
e.printStackTrace(); }
            }
            System.out.println("Lambda Thread finished.");
        });
        t3.start();
    }
}
```

### Thread synchronization techniques

When multiple threads access and modify shared resources (data or objects), it can lead to data inconsistency and race conditions. Thread synchronization ensures that only one thread accesses a critical section (shared resource) at a time.

- `synchronized` **keyword:**
  - `synchronized` **method:** When applied to a method, it locks the object (for instance methods) or the class (for static methods). Only one `synchronized` method can execute on that object (or class) at a time.
  - `synchronized` **block:** Allows you to synchronize on a specific object (monitor) and only protect a specific block of code, giving more fine-grained control than a synchronized method.

    ```
    class Counter {
        int count = 0;
        // Synchronized method
        public synchronized void increment() {
            count++;
        }
        // Synchronized block
        public void decrement() {
            synchronized (this) { // Locks on the current object
                count--;
            }
        }
        // For static methods, locks on the Class object
        public static synchronized void staticIncrement() { /* ... */ }
    }
    ```

- `wait()`, `notify()`, `notifyAll()` **methods (from** `Object` **class):**
  - **Purpose:** For inter-thread communication. They must be called within a `synchronized` block/method.
  - `wait()`: Causes the current thread to release the lock and go into a waiting state until another thread invokes `notify()` or `notifyAll()` on the same object.
  - `notify()`: Wakes up a single waiting thread.
  - `notifyAll()`: Wakes up all waiting threads.
  - **Use Cases:** Producer-Consumer problem, thread coordination.
- `java.util.concurrent` **package (High-level concurrency utilities):**
  - Provides more powerful, flexible, and often more performant alternatives to low-level `synchronized` and `wait/notify`.
  - `Lock` **interface (**`ReentrantLock`**):** Explicit locking mechanism.
  - `Semaphore`**:** Controls access to a limited number of resources.
  - `CountDownLatch`**,** `CyclicBarrier`**:** For thread synchronization points.
  - `ExecutorService` **/** `ThreadPoolExecutor`**:** For managing thread pools.
  - `Concurrent Collections` **(**`ConcurrentHashMap`**,** `CopyOnWriteArrayList`**):** Thread-safe collections that offer better scalability than synchronized wrapper collections.

### Applying thread safety to Collection framework classes.

- **Legacy Thread-Safe Collections:**
  - `Vector`: A legacy class, thread-safe version of `ArrayList`.
  - `Hashtable`: A legacy class, thread-safe version of `HashMap`.

- **Synchronized Wrappers:** `Collections` class provides static factory methods to create synchronized (thread-safe) versions of existing collection objects.

  ```
  List<String> syncList = Collections.synchronizedList(new ArrayList<>());
  Map<String, String> syncMap = Collections.synchronizedMap(new HashMap<>());
  Set<String> syncSet = Collections.synchronizedSet(new HashSet<>());
  ```

  - **Limitation:** These wrappers achieve thread safety by synchronizing on the entire collection object for every method call. This can become a performance bottleneck for high concurrency.
- **Concurrent Collections (`java.util.concurrent`):**
  - **Purpose:** Designed for concurrent access, offering better scalability and performance than synchronized wrappers by using more sophisticated locking mechanisms (e.g., fine-grained locking, lock-free algorithms).
  - **Examples:**
    - `ConcurrentHashMap`: Highly concurrent map.
    - `CopyOnWriteArrayList`: A thread-safe `List` useful when reads vastly outnumber writes.
    - `ConcurrentLinkedQueue`: A thread-safe unbounded queue.
    - `BlockingQueue` implementations (`ArrayBlockingQueue`, `LinkedBlockingQueue`): For producer-consumer scenarios.
  - **Recommendation:** Prefer Concurrent Collections over synchronized wrappers or legacy synchronized collections (`Vector`, `Hashtable`) for concurrent scenarios.

---

## Database Access Methods, JDBC, driver & architecture

**Database Access:** Refers to how Java applications connect to and interact with relational databases.

### JDBC (Java Database Connectivity)

- **Definition:** JDBC is a Java API that provides a standard way for Java applications to connect to, query, and update relational databases. It's a specification, not an implementation.
- **Purpose:** Abstracts away the database-specific details, allowing developers to write generic database code that can work with different databases by just changing the JDBC driver.
- **Components:**
  1. **JDBC API:** The set of Java interfaces and classes (in `java.sql` and `javax.sql` packages) provided to application developers.
  2. **JDBC Driver:** A software component (provided by the database vendor) that implements the JDBC API and translates JDBC calls into the database's native communication protocol.
  3. **Database:** The actual relational database management system (RDBMS) where the data resides.

### Driver & Architecture (JDBC Architecture)

The JDBC architecture consists of two layers:

1. **JDBC API:** This is the `java.sql` API that Java applications use to issue SQL commands and process results.

2. **JDBC Driver Manager:** This is the `java.sql.DriverManager` class. It loads JDBC drivers, establishes connections to databases, and manages the set of registered drivers.
3. **JDBC Driver:** This is a set of classes that implement the JDBC interfaces for a specific database. Database vendors provide these drivers.

**Types of JDBC Drivers (legacy classification, still relevant conceptually):**

- **Type 1: JDBC-ODBC Bridge Driver:** (Deprecated/Legacy) Translates JDBC calls into ODBC calls, which then connect to the database. Requires ODBC driver installation. Slow and platform-dependent.
- **Type 2: Native-API Driver (Partially Java Driver):** Converts JDBC calls into the native API calls of the database. Requires native libraries on the client machine. Faster than Type 1 but still platform-dependent.
- **Type 3: Network Protocol Driver (Middleware Driver):** Uses a middleware server to translate JDBC calls into a database-specific network protocol. Fully Java, allows clients to connect to different databases via the middleware.
- **Type 4: Native-Protocol Pure Java Driver:** (Most common and recommended) Converts JDBC calls directly into the database's native network protocol. Written entirely in Java, highly portable, fast, and does not require native libraries.

### The `java.sql` package

This package provides the core interfaces and classes for JDBC API.

#### `Driver Manager`, `Connection`, `Statement`, `PreparedStatement`, `Result Set`

These are the fundamental JDBC objects for database interaction.

1. **`DriverManager`:**
   - **Purpose:** Manages the JDBC drivers. It's the first step in connecting to a database.
   - **Key Method:** `getConnection(String url, String user, String password)`: Establishes a connection to the database. The `url` contains the database type, location, and name (e.g., `jdbc:mysql://localhost:3306/mydb`).
2. **`Connection`:**
   - **Purpose:** Represents an active connection to a specific database. All communication with the database happens through this object.
   - **Responsibility:** Manages transactions (commit, rollback), creates `Statement` objects.
   - **Crucial:** Always close the `Connection` when done using `connection.close()` (preferably in a `finally` block or `try-with-resources`).
3. **`Statement`:**
   - **Purpose:** Used to execute simple, static SQL queries (without parameters).
   - **Methods:** `executeQuery(String sql)` (for `SELECT`), `executeUpdate(String sql)` (for `INSERT`, `UPDATE`, `DELETE`).
4. **`PreparedStatement`:**
   - **Purpose:** (RECOMMENDED) Used to execute pre-compiled SQL queries with parameters.
   - **Benefits:**
     - **Performance:** SQL query is parsed and optimized by the database once, then reused.
     - **Security:** Prevents SQL Injection attacks by safely binding parameters.
     - **Readability:** Cleaner code for queries with many parameters.

   - **Methods:** `setXxx(int parameterIndex, value)` to set parameters, then `executeQuery()` or `executeUpdate()`.
5. `ResultSet`:
   - **Purpose:** Represents the tabular data returned by a `SELECT` query.
   - **Cursor:** Acts as an iterator, initially positioned before the first row.
   - **Methods:** `next()` (moves to the next row), `getXxx(int columnIndex)` or `getXxx(String columnName)` to retrieve data by type.
   - **Crucial:** Always close the `ResultSet` when done.

**Typical JDBC Workflow:**

1. **Load Driver (Optional for modern JDBC):** `Class.forName("com.mysql.cj.jdbc.Driver");` (modern JDBC 4.0+ drivers auto-register).
2. **Establish Connection:** `Connection conn = DriverManager.getConnection(url, user, password);`
3. **Create Statement (or PreparedStatement):** `Statement stmt = conn.createStatement();` or `PreparedStatement pstmt = conn.prepareStatement(sql);`
4. **Execute Query:** `ResultSet rs = stmt.executeQuery(sql);` or `int count = pstmt.executeUpdate();`
5. **Process Result (if SELECT):** Iterate through `ResultSet`.
6. **Close Resources:** `rs.close()`, `stmt.close()` (or `pstmt.close()`), `conn.close()`. **Crucial to use `try-with-resources` for automatic closing.**

**Example (Using `PreparedStatement` and `try-with-resources`):**

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JdbcDemo {

    private static final String JDBC_URL = "jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1";
// In-memory H2 database
    private static final String USER = "sa";
    private static final String PASSWORD = "";

    public static void main(String[] args) {
        // Step 1: Create Table (or connect to existing DB)
        try (Connection conn = DriverManager.getConnection(JDBC_URL, USER,
PASSWORD);
             Statement stmt = conn.createStatement()) {
            String createTableSQL = "CREATE TABLE IF NOT EXISTS users (id INT
PRIMARY KEY AUTO_INCREMENT, name VARCHAR(255), email VARCHAR(255))";
            stmt.executeUpdate(createTableSQL);
            System.out.println("Table 'users' created or already exists.");
        } catch (SQLException e) {
            e.printStackTrace();
        }
        // Step 2: Insert Data
        insertUser("Alice", "alice@example.com");
        insertUser("Bob", "bob@example.com");

        // Step 3: Select Data
        System.out.println("\nAll Users:");
        selectAllUsers();

        // Step 4: Update Data
        updateUser(1, "Alice Smith", "alice.smith@example.com");

        // Step 5: Select Data Again
        System.out.println("\nUsers after update:");
        selectAllUsers();

        // Step 6: Delete Data
        deleteUser(2);

        // Step 7: Select Data Again
        System.out.println("\nUsers after delete:");
        selectAllUsers();
    }

    private static void insertUser(String name, String email) {
        String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
        try (Connection conn = DriverManager.getConnection(JDBC_URL, USER,
PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setString(1, name);
            pstmt.setString(2, email);
            int rowsAffected = pstmt.executeUpdate();
            System.out.println("Inserted " + rowsAffected + " user(s).");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void selectAllUsers() {
        String sql = "SELECT id, name, email FROM users";
        try (Connection conn = DriverManager.getConnection(JDBC_URL, USER,
PASSWORD);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                String email = rs.getString("email");
                System.out.println("ID: " + id + ", Name: " + name + ", Email: " +
email);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void updateUser(int id, String newName, String newEmail) {
        String sql = "UPDATE users SET name = ?, email = ? WHERE id = ?";
        try (Connection conn = DriverManager.getConnection(JDBC_URL, USER,
PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setString(1, newName);
            pstmt.setString(2, newEmail);
            pstmt.setInt(3, id);
            int rowsAffected = pstmt.executeUpdate();
            System.out.println("Updated " + rowsAffected + " user(s).");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void deleteUser(int id) {
        String sql = "DELETE FROM users WHERE id = ?";
        try (Connection conn = DriverManager.getConnection(JDBC_URL, USER,
PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setInt(1, id);
            int rowsAffected = pstmt.executeUpdate();
            System.out.println("Deleted " + rowsAffected + " user(s).");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

---

## `CallableStatement`, Stored procedure & functions

### `CallableStatement`

- **Purpose:** Used to execute SQL stored procedures and functions in a database.
- **Benefits:**
    - **Performance:** Stored procedures are pre-compiled on the database server.
    - **Security:** Reduces SQL injection risk when parameters are properly bound.
    - **Encapsulation:** Business logic can be encapsulated on the database side.
    - **Network Traffic:** Can reduce network traffic by executing complex logic on the server.
- **Methods:**
    - `prepareCall(String sql)`: Creates a `CallableStatement` object (SQL string typically uses `"{call procedure_name(?, ?)}"`, `"{?= call function_name(?, ?)}"`, etc.).
    - `setXxx(int parameterIndex, value)`: Sets input parameters.
    - `registerOutParameter(int parameterIndex, int sqlType)`: Registers output parameters (including return values of functions).
    - `execute()`: Executes the procedure/function.

- `getXxx(int parameterIndex)`: Retrieves output parameter values.

**Example (Calling a Stored Procedure - simplified):**

Assume a stored procedure `getUserCount(OUT count INT)`:

```
-- Example MySQL Stored Procedure
DELIMITER //
CREATE PROCEDURE getUserCount(OUT user_count INT)
BEGIN
    SELECT COUNT(*) INTO user_count FROM users;
END //
DELIMITER ;
```

```
import java.sql.CallableStatement;

public class CallableStatementDemo {
    public static void getUserCount() {
        String sql = "{CALL getUserCount(?)}"; // SQL to call procedure with one
OUT parameter
        try (Connection conn = DriverManager.getConnection(JDBC_URL, USER,
PASSWORD);
             CallableStatement cstmt = conn.prepareCall(sql)) {

            cstmt.registerOutParameter(1, java.sql.Types.INTEGER); // Register
output parameter at index 1
            cstmt.execute(); // Execute the procedure

            int userCount = cstmt.getInt(1); // Retrieve the output value
            System.out.println("Total users: " + userCount);

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

**Example (Calling a Stored Function - simplified):**

Assume a stored function `getUserName(IN userId INT) RETURNS VARCHAR(255)`:

```
-- Example MySQL Stored Function
DELIMITER //
CREATE FUNCTION getUserName(user_id INT) RETURNS VARCHAR(255)
BEGIN
    DECLARE user_name VARCHAR(255);
    SELECT name INTO user_name FROM users WHERE id = user_id;
    RETURN user_name;
```

```
END //
DELIMITER ;
```

```
public class CallableFunctionDemo {
    public static void getUserNameById(int userId) {
        String sql = "{? = CALL getUserName(?)}"; // SQL to call function with
return value and one IN parameter
        try (Connection conn = DriverManager.getConnection(JDBC_URL, USER,
PASSWORD);
             CallableStatement cstmt = conn.prepareCall(sql)) {

            cstmt.registerOutParameter(1, java.sql.Types.VARCHAR); // Register
return value at index 1
            cstmt.setInt(2, userId); // Set input parameter at index 2
            cstmt.execute(); // Execute the function

            String userName = cstmt.getString(1); // Retrieve the return value
            System.out.println("User Name for ID " + userId + ": " + userName);

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

### Scrollable `ResultSet`

- **Default `ResultSet`:** By default, `ResultSet` objects are `TYPE_FORWARD_ONLY`, meaning you can only move the cursor forward (`next()`).
- **Scrollable `ResultSet`:** Allows you to move the cursor both forward and backward, and even to a specific row.
- **Types:**
  - **`TYPE_SCROLL_INSENSITIVE`:** Can move cursor forward and backward. Not sensitive to changes made by other transactions after the `ResultSet` was created.
  - **`TYPE_SCROLL_SENSITIVE`:** Can move cursor forward and backward. Sensitive to changes made by others (might reflect latest data).
- **Creating a Scrollable `ResultSet`:** Specify `ResultSet.TYPE_SCROLL_INSENSITIVE` or `TYPE_SCROLL_SENSITIVE` when creating the `Statement` or `PreparedStatement`.
- **Concurrency:** Also possible to specify `ResultSet.CONCUR_READ_ONLY` (default) or `CONCUR_UPDATABLE` (allows updating/deleting rows via `ResultSet`).

**Example:**

```
import java.sql.ResultSet;
import java.sql.Statement;

public class ScrollableResultSetDemo {
    public static void demonstrateScrollableResultSet() {
        String sql = "SELECT id, name FROM users";
        try (Connection conn = DriverManager.getConnection(JDBC_URL, USER,
PASSWORD);
             Statement stmt = conn.createStatement(
                 ResultSet.TYPE_SCROLL_INSENSITIVE, // Make it scrollable and
insensitive to changes
                 ResultSet.CONCUR_READ_ONLY)) {      // Read-only

            ResultSet rs = stmt.executeQuery(sql);

            // Move to last row
            if (rs.last()) {
                System.out.println("Last user: " + rs.getString("name"));
            }

            // Move to first row
            if (rs.first()) {
                System.out.println("First user: " + rs.getString("name"));
            }

            // Move relative
            if (rs.absolute(2)) { // Move to 2nd row
                System.out.println("Second user: " + rs.getString("name"));
            }

            if (rs.previous()) { // Move to previous row (1st row)
                System.out.println("Previous user: " + rs.getString("name"));
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

### Writing multi-tiered DB applications (Use DAO & DTO layers)

Multi-tiered (or N-tier) architecture separates an application into logical and physical layers, promoting modularity, scalability, and maintainability. For database applications, common tiers are:

1. **Presentation Layer (UI/Web):**
   - Handles user interaction, displays data, sends user input.
   - (e.g., Servlet/JSP, Spring MVC Controller, REST Controller).
2. **Service Layer (Business Logic Layer):**
   - Contains core business rules and logic.
   - Orchestrates operations across multiple DAOs to fulfill business transactions.
   - Manages transaction boundaries.
   - (e.g., Spring `@Service` class).
3. **Data Access Layer (DAO Layer):**

   - **DAO (Data Access Object):** An architectural pattern used to isolate application (business) logic from persistence technologies (e.g., JDBC, JPA, Hibernate).
   - **Purpose:** Provides a consistent API for accessing data regardless of the underlying database or persistence mechanism.
   - **Responsibilities:** Performs CRUD operations, maps database results to Java objects.
   - (e.g., JDBC code directly, Spring Data JPA Repository).
4. **Database Layer:**
   - The actual database (e.g., MySQL, PostgreSQL, Oracle).

**DTO (Data Transfer Object):**

- **Purpose:** A simple POJO (Plain Old Java Object) used to transfer data between different layers of an application.
- **Characteristics:** Contains only fields, constructors, getters, and setters. No business logic.
- **Benefits:**
  - **Reduced Network Traffic:** Can bundle multiple pieces of data into a single object for transfer.
  - **Decoupling:** Decouples the presentation layer from the internal domain model/entity objects, providing a clear contract for data exchange. This is especially important for REST APIs where you might not want to expose your full JPA entities.

**Example Multi-tiered Structure:**

```java
// 1. Data Transfer Object (DTO) - Used for communication between layers
public class UserDTO {
    private int id;
    private String name;
    private String email;

    // Constructors, Getters, Setters
}

// 2. Data Access Object (DAO) Layer - Responsible for database interaction
interface UserDAO {
    UserDTO findById(int id);
    List<UserDTO> findAll();
    int save(UserDTO user); // Returns ID or rows affected
    int update(UserDTO user);
    int delete(int id);
}

// UserDAOImpl.java (Implementation using JDBC)
public class UserDAOImpl implements UserDAO {
    // JDBC connection, PreparedStatement, ResultSet code here
    // Maps ResultSet rows to UserDTO objects
    // Maps UserDTO objects to PreparedStatement parameters
}

// 3. Service Layer - Contains business logic, orchestrates DAOs
public class UserService {
    private UserDAO userDAO; // Injected via constructor or setter
    public UserService(UserDAO userDAO) {
        this.userDAO = userDAO;
    }

    // Business method that might involve multiple DAO calls or validations
    public UserDTO registerUser(UserDTO user) {
        // Perform business validation
        if (user.getName() == null || user.getName().isEmpty()) {
            throw new IllegalArgumentException("User name cannot be empty.");
        }
        // Save to DB via DAO
        int newId = userDAO.save(user);
        user.setId(newId); // Set ID if generated by DB
        return user;
    }

    public List<UserDTO> getAllUsers() {
        return userDAO.findAll();
    }
}

// 4. Presentation Layer (e.g., a simple Console App or Web Controller)
public class UserConsoleApp {
    private UserService userService;

    public UserConsoleApp(UserService userService) {
        this.userService = userService;
    }

    public void run() {
        UserDTO newUser = new UserDTO();
        newUser.setName("Charlie");
        newUser.setEmail("charlie@example.com");

        try {
            UserDTO registeredUser = userService.registerUser(newUser);
            System.out.println("Registered user: " + registeredUser.getName());
        } catch (IllegalArgumentException e) {
            System.err.println("Error: " + e.getMessage());
        }

        List<UserDTO> allUsers = userService.getAllUsers();
        System.out.println("All users:");
        for (UserDTO user : allUsers) {
            System.out.println("- " + user.getName() + " (" + user.getEmail() +
")");
        }
    }

    public static void main(String[] args) {
        // Manual dependency injection for demonstration
        UserDAO userDAO = new UserDAOImpl(); // Needs actual JDBC implementation
        UserService userService = new UserService(userDAO);
        UserConsoleApp app = new UserConsoleApp(userService);
        app.run();
    }
}
```

---

## Java 8 Interfaces, default methods (Recap from Session 8)

- **Default Methods:** Methods in an interface with an implementation (`default` keyword). Allow adding new methods to interfaces without breaking existing implementations.
- **Static Methods:** Methods in an interface with an implementation (`static` keyword). Utility methods related to the interface.

## Lambda expressions

**Lambda Expressions:** A new feature in Java 8 that provides a concise way to represent an instance of a functional interface (an interface with exactly one abstract method). They enable functional programming styles in Java.

- **Syntax:** `(parameters) -> expression` or `(parameters) -> { statements; }`
- **Implicit Typing:** Type inference often allows omitting parameter types.
- **Return Value:** If the body is a single expression, its value is automatically returned.
- **Use Cases:** Short, inline implementations of functional interfaces, especially for callbacks, event listeners, and stream operations.

**Example:**

```java
// Old way (Anonymous Inner Class)
Runnable oldRunnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running old way.");
    }
};

// New way (Lambda Expression for Runnable)
Runnable lambdaRunnable = () -> System.out.println("Running with Lambda.");

new Thread(oldRunnable).start();
new Thread(lambdaRunnable).start();


// Lambda with parameters and return value (for a custom functional interface)
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}

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

```
System.out.println("10 + 5 = " + addition.operate(10, 5));
System.out.println("10 - 5 = " + subtraction.operate(10, 5));
```

## Functional interfaces, Built-in functional interfaces

- **Functional Interface:** An interface with exactly one abstract method. (Covered in Session 8)
- **Built-in Functional Interfaces (from `java.util.function` package):** Java 8 introduced several standard functional interfaces for common patterns.
  - **`Predicate<T>`:** Represents a boolean-valued function of one argument. (`test(T)`).
    - Example: `Predicate<Integer> isEven = num -> num % 2 == 0;`
  - **`Consumer<T>`:** Represents an operation that accepts a single input argument and returns no result. (`accept(T)`).
    - Example: `Consumer<String> printMessage = msg -> System.out.println(msg);`
  - **`Supplier<T>`:** Represents a supplier of results. Takes no arguments and returns a value. (`get()`).
    - Example: `Supplier<Double> randomValue = () -> Math.random();`
  - **`Function<T, R>`:** Represents a function that accepts one argument and produces a result. (`apply(T)`).
    - Example: `Function<String, Integer> stringLength = s -> s.length();`
  - **`BiPredicate<T, U>`, `BiConsumer<T, U>`, `BiFunction<T, U, R>`:** For two arguments.
  - **UnaryOperator (extends Function<T,T>)**: Takes one argument, returns same type.
  - **BinaryOperator (extends BiFunction<T,T,T>)**: Takes two arguments, returns same type.
  - And many primitive specialized versions (e.g., `IntPredicate`, `LongConsumer`, `DoubleFunction`).

## Method and Constructor references.

Method and constructor references provide an even more compact syntax for lambda expressions that simply call an existing method or constructor.

- **Method References:**

  - **Static method reference:** `ClassName::staticMethodName`

  - **Instance method reference (on a particular object):** `objectName::instanceMethodName`

  - **Instance method reference (on an arbitrary object of a particular type):** `ClassName::instanceMethodName`

  - **Example:**

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

    // Lambda: name -> System.out.println(name)
    names.forEach(System.out::println); // Method reference to static
    method

    // Lambda: s -> s.length()
    List<Integer> lengths = names.stream()
                                 .map(String::length) // Method reference to
    ```

```
instance method
                             .collect(Collectors.toList());
```

- **Constructor References:**
  - **Syntax:** `ClassName::new`
  - **Example:**

    ```
    // Assuming a Person class with a constructor Person(String name)
    // Lambda: name -> new Person(name)
    Function<String, Person> personCreator = Person::new; // Constructor reference
    Person newPerson = personCreator.apply("David");
    ```

## `java.util.Stream`, stream operations & executions.

**Stream API:** A powerful new API in Java 8 for processing collections of objects (and other data sources) in a functional and declarative style.

- **Concept:** A sequence of elements that supports sequential and parallel aggregate operations. Streams do not store data; they operate on a source (like a `List`, `Set`, `Map`, array, or I/O channel).
- **Characteristics:**
  - **Declarative:** You describe *what* you want to do, not *how* to do it.
  - **Functional:** Operations are applied to elements.
  - **Immutable Source:** Streams do not modify the underlying data source.
  - **Lazy Evaluation:** Operations are not performed until a terminal operation is invoked.
  - **Pipelines:** Operations can be chained together to form a pipeline.
- **Stream Operations:**
  1. **Source:** From where the stream originates (e.g., `list.stream()`, `Arrays.stream(array)`).
  2. **Intermediate Operations:** Transform a stream into another stream. They are lazy and return a new `Stream`. Can be chained.
     - `filter(Predicate<T>)`: Selects elements matching a condition.
     - `map(Function<T, R>)`: Transforms elements from one type to another.
     - `flatMap(Function<T, Stream<R>>)`: Transforms each element into a stream of elements, then flattens these streams into a single stream.
     - `distinct()`: Returns a stream with unique elements.
     - `sorted()` / `sorted(Comparator<T>)`: Sorts elements.
     - `limit(long maxSize)`: Truncates the stream to a maximum size.
     - `skip(long n)`: Skips the first `n` elements.
  3. **Terminal Operations:** Produce a result or a side-effect. They consume the stream and mark the end of the pipeline.
     - **Collection:** `collect(Collector)`: Gathers elements into a collection (e.g., `Collectors.toList()`, `Collectors.toSet()`, `Collectors.toMap()`).

- **Iteration:** `forEach(Consumer<T>)`: Performs an action for each element.
- **Reduction:** `reduce()`, `sum()`, `min()`, `max()`, `count()`.
- **Matching:** `anyMatch()`, `allMatch()`, `noneMatch()`.
- **Finding:** `findFirst()`, `findAny()`.

**Example:**

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

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

        // Filter names starting with 'A', map to uppercase, sort, then collect to
a new list
        List<String> filteredSortedNames = names.stream() // Source
            .filter(name -> name.startsWith("A")) // Intermediate operation
            .map(String::toUpperCase) // Intermediate operation
            .sorted() // Intermediate operation
            .collect(Collectors.toList()); // Terminal operation

        System.out.println("Filtered and Sorted: " + filteredSortedNames); //
[ALICE, ANNA, ANDREW] (depends on sorting order)

        // Count names with specific length
        long count = names.stream()
            .filter(name -> name.length() > 4)
            .count(); // Terminal operation
        System.out.println("Names longer than 4 chars: " + count);

        // Find the first name starting with 'B'
        names.stream()
            .filter(name -> name.startsWith("B"))
            .findFirst() // Terminal operation, returns Optional
            .ifPresent(name -> System.out.println("Found: " + name));

        // Sum of lengths of names
        int totalLength = names.stream()
            .mapToInt(String::length) // map to IntStream
            .sum(); // Terminal operation
        System.out.println("Total length of all names: " + totalLength);
    }
}
```

---

## Advanced `java.util.Stream` operations (e.g. `reduce()`, `flatMap()`, etc).

### `reduce()`

- **Purpose:** A terminal operation that combines the elements of a stream into a single result using a given binary operator. It's useful for aggregating values.
- **Variants:**
  1. `T reduce(T identity, BinaryOperator<T> accumulator)`:
     - `identity`: Initial value of the reduction.
     - `accumulator`: A function that combines a partial result with the next element.
  2. `Optional<T> reduce(BinaryOperator<T> accumulator)`: Returns an `Optional` because there might be no elements to reduce.
  3. `<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`: For parallel streams to combine partial results.

**Example:**

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

// Sum of numbers (using identity)
int sum = numbers.stream()
                 .reduce(0, (acc, num) -> acc + num);
System.out.println("Sum: " + sum); // Output: 15

// Product of numbers (using Optional)
Optional<Integer> product = numbers.stream()
                                   .reduce((acc, num) -> acc * num);
product.ifPresent(p -> System.out.println("Product: " + p)); // Output: 120

// Concatenate strings
List<String> words = Arrays.asList("Java", "Stream", "API");
String sentence = words.stream()
                       .reduce("", (acc, word) -> acc + " " + word)
                       .trim(); // Trim leading space
System.out.println("Sentence: " + sentence); // Output: Java Stream API
```

### `flatMap()`

- **Purpose:** An intermediate operation that transforms each element of a stream into a stream of zero or more elements, and then concatenates (flattens) all these resulting streams into a single new stream.
- **Use Cases:** When you have a collection of collections (or objects that can produce multiple elements), and you want to process all elements from all inner collections as a single stream.

**Example:**

```java
List<List<String>> listOfLists = Arrays.asList(
    Arrays.asList("apple", "banana"),
    Arrays.asList("orange", "grape", "kiwi"),
    Arrays.asList("melon")
);
```

```java
// Get all fruits into a single stream
List<String> allFruits = listOfLists.stream()
                                    .flatMap(List::stream) // FlatMap each inner
list into a stream and flatten
                                    .collect(Collectors.toList());
System.out.println("All fruits: " + allFruits);
// Output: [apple, banana, orange, grape, kiwi, melon]

// Example with objects: Get all unique skills from a list of employees
class Employee {
    String name;
    List<String> skills;
    Employee(String n, List<String> s) { name = n; skills = s; }
    public List<String> getSkills() { return skills; }
}

List<Employee> employees = Arrays.asList(
    new Employee("Alice", Arrays.asList("Java", "SQL")),
    new Employee("Bob", Arrays.asList("Python", "Java", "AWS")),
    new Employee("Charlie", Arrays.asList("SQL", "Azure"))
);

List<String> uniqueSkills = employees.stream()
                                     .flatMap(e -> e.getSkills().stream()) //
Stream of skills for each employee
                                     .distinct() // Get unique skills
                                     .sorted()
                                     .collect(Collectors.toList());
System.out.println("Unique skills: " + uniqueSkills);
// Output: [AWS, Azure, Java, Python, SQL]
```

### Reflection concepts

**Reflection:** A powerful feature in Java that allows a running Java application to inspect (examine) and modify its own structure, behavior, and members (classes, methods, fields, constructors) at runtime.

- **`java.lang.reflect` package:** Provides the core classes for reflection (`Class`, `Method`, `Field`, `Constructor`).
- **`Class` object:** The entry point for reflection. You can get a `Class` object for any type using `ClassName.class`, `object.getClass()`, or `Class.forName("fully.qualified.ClassName")`.

### Invoking methods dynamically, Assigning values to private fields.

Reflection allows you to do things that are normally prevented by access modifiers or static typing.

- **Invoking methods dynamically:**
  - You can get a `Method` object for a method by its name and parameter types.
  - Then invoke it using `method.invoke(object, args)`.
  - Can invoke private methods by first calling `method.setAccessible(true)`.

**Example:**

```java
import java.lang.reflect.Method;

class ReflectionTarget {
    public void publicMethod(String msg) {
        System.out.println("Public method called: " + msg);
    }
    private void privateMethod(int value) {
        System.out.println("Private method called with value: " + value);
    }
}

public class DynamicMethodInvocation {
    public static void main(String[] args) throws Exception {
        ReflectionTarget target = new ReflectionTarget();
        Class<?> cls = target.getClass(); // Get Class object

        // Invoke public method dynamically
        Method publicMethod = cls.getMethod("publicMethod", String.class);
        publicMethod.invoke(target, "Hello from Reflection!");

        // Invoke private method dynamically
        Method privateMethod = cls.getDeclaredMethod("privateMethod", int.class);
        privateMethod.setAccessible(true); // Allow access to private method
        privateMethod.invoke(target, 42);
    }
}
```

- **Assigning values to private fields dynamically:**
  - You can get a `Field` object for a field by its name.
  - Set its value using `field.set(object, value)`.
  - Can modify private fields by first calling `field.setAccessible(true)`.

**Example:**

```java
import java.lang.reflect.Field;

class Person {
    private String name;
    public Person(String name) { this.name = name; }
    public String getName() { return name; }
}

public class PrivateFieldManipulation {
    public static void main(String[] args) throws Exception {
        Person person = new Person("Original Name");
        System.out.println("Initial Name: " + person.getName());

        Class<?> cls = person.getClass();
        Field nameField = cls.getDeclaredField("name"); // Get Field object for
'name'
```

```
        nameField.setAccessible(true); // Allow access to private field
        nameField.set(person, "Changed Name via Reflection"); // Set value

        System.out.println("Modified Name: " + person.getName());
    }
}
```

**Caution:** Reflection should be used sparingly as it breaks encapsulation and type safety, can be slower, and makes code harder to read and maintain. It's often used in frameworks (e.g., Spring, ORMs) but less frequently in application code.

### Annotation concept, retention policies

**Annotations:** Metadata that can be added to Java code elements (classes, methods, fields, parameters, etc.) to provide information to the compiler, runtime, or other tools. They don't directly affect code execution but provide supplementary data.

- **Declaration:** `public @interface MyAnnotation { ... }`
- **Applying Annotations:** `@Override`, `@Deprecated`, `@SuppressWarnings`, custom annotations.

**Meta-Annotations:** Annotations that annotate other annotations.

- **`@Retention`:**
  - **Purpose:** Specifies how long the annotation will be retained.
  - **Retention Policies:**
    - **`RetentionPolicy.SOURCE`:** Annotation is retained only in the source code (`.java` file) and is discarded by the compiler. (e.g., `@Override`).
    - **`RetentionPolicy.CLASS`:** Annotation is retained in the `.class` file during compilation but is not available at runtime via reflection. (Default policy).
    - **`RetentionPolicy.RUNTIME`:** Annotation is retained in the `.class` file and is available at runtime via reflection. (Required for annotations processed by frameworks like Spring, JUnit).
- **`@Target`:** Specifies the type of program element to which an annotation can be applied (e.g., `ElementType.TYPE`, `ElementType.METHOD`, `ElementType.FIELD`).
- **`@Inherited`:** Indicates that an annotation type is automatically inherited by subclasses.
- **`@Documented`:** Indicates that an annotation should be documented by Javadoc.

**Example (Custom Annotation with Retention Policy):**

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

// Annotation to mark a method as a "Logger" action
@Retention(RetentionPolicy.RUNTIME) // Retain at runtime for reflection processing
@Target(ElementType.METHOD) // Can only be applied to methods
public @interface Loggable {
    String value() default "Default Action"; // An element with a default value
}

// Example usage
class Service {
    @Loggable("Performing business logic")
    public void doBusinessLogic() {
        System.out.println("Business logic executed.");
    }

    @Loggable // Using default value
    public void anotherMethod() {
        System.out.println("Another method executed.");
    }
}

// Runtime processing of annotation
public class AnnotationProcessor {
    public static void main(String[] args) {
        Service service = new Service();
        for (Method method : service.getClass().getDeclaredMethods()) {
            if (method.isAnnotationPresent(Loggable.class)) {
                Loggable loggableAnnotation =
method.getAnnotation(Loggable.class);
                System.out.println("Method '" + method.getName() + "' is loggable.
Action: " + loggableAnnotation.value());
            }
        }
    }
}
```