The heart of Java's power: Object-Oriented Programming (OOP)

Classes & Objects

Process vs Object-Oriented

Understanding the shift from process-oriented to object-oriented programming is key to grasping Java's philosophy.

- **Process-Oriented Programming (POP):**
    - Focuses on a sequence of steps or functions to solve a problem.
    - Data and functions are separate. Functions often operate on global data.
    - Examples: C, Pascal.
    - **Analogy:** Imagine a factory assembly line. You have distinct steps (functions) like "cut wood," "assemble parts," "paint," and the raw materials (data) flow through these steps. If you change a step, you might need to change many places where that data is processed.
- **Object-Oriented Programming (OOP):**
    - Focuses on "objects" which are instances of "classes."
    - An object bundles **data (attributes/fields)** and the **functions (methods)** that operate on that data into a single unit. This is called **encapsulation**.
    - Key principles: Encapsulation, Inheritance, Polymorphism, Abstraction.
    - Examples: Java, C++, Python.
    - **Analogy:** Imagine a car. It's an object with attributes (color, speed, fuel level) and behaviors (accelerate, brake, turn). These attributes and behaviors are tightly coupled within the car object. If you want to change the car's speed, you use its `accelerate()` or `brake()` methods, not some global "changeSpeed" function. This makes code more modular, maintainable, and reusable.

Instance Variables and Methods

When you define a class, you're creating a blueprint. **Instance variables** and **instance methods** are the data and behavior *specific to each object* created from that blueprint.

- **Instance Variables (Non-Static Fields):**
    - These are variables declared inside a class but outside any method, constructor, or block.
    - Each object (instance) of the class gets its *own copy* of these variables.
    - They store the unique state or attributes of an object.
    - Example: `color`, `speed`, `fuelLevel` for a `Car` object.
- **Instance Methods (Non-Static Methods):**
    - These are methods defined within a class that operate on the instance variables of a specific object.

    - They define the behaviors or actions that an object can perform.
    - To call an instance method, you need an object of that class.
    - Example: `accelerate()`, `brake()`, `getFuelLevel()` for a `Car` object.

Example:

```java
class Dog {
    // Instance Variables
    String name;
    String breed;
    int age;

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

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

public class InstanceMembersExample {
    public static void main(String[] args) {
        // Creating the first Dog object (instance)
        Dog myDog = new Dog(); // myDog is an object/instance
        myDog.name = "Buddy";    // Setting instance variables for myDog
        myDog.breed = "Golden Retriever";
        myDog.age = 3;

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

        // Creating a second Dog object (instance)
        Dog sisterDog = new Dog(); // sisterDog is another object/instance
        sisterDog.name = "Lucy";
        sisterDog.breed = "Labrador";
        sisterDog.age = 2;

        // Note that sisterDog has its own separate copies of name, breed, age
        sisterDog.bark();
        sisterDog.displayInfo();
    }
}
```

Declaring and Using Objects

To use an object, you typically follow these steps:

1. **Declaration:** Declare a variable of the class type. This variable will hold a reference to the object. `ClassName objectName;`

2. **Instantiation:** Create an actual object using the `new` keyword and a constructor. This allocates memory for the object. `objectName = new ClassName();`
3. **Initialization:** Use the object variable (reference) to access its instance variables and methods.

Example:

```java
class Car {
    String color;
    int speed;

    void startEngine() {
        System.out.println(color + " car engine started.");
    }

    void accelerate(int increase) {
        speed += increase;
        System.out.println("Car accelerated to " + speed + " mph.");
    }
}

public class DeclaringAndUsingObjects {
    public static void main(String[] args) {
        // 1. Declaration: Declares a reference variable 'myCar' of type Car
        Car myCar;

        // 2. Instantiation: Creates a new Car object in memory
        //    and assigns its reference to 'myCar'
        myCar = new Car();

        // 3. Initialization/Usage: Accessing instance variables and methods
        myCar.color = "Red"; // Assigning value to an instance variable
        myCar.speed = 0;

        myCar.startEngine(); // Calling an instance method
        myCar.accelerate(50); // Calling an instance method with an argument

        System.out.println("My car's color: " + myCar.color);
        System.out.println("My car's current speed: " + myCar.speed);

        // You can also combine declaration and instantiation:
        Car anotherCar = new Car();
        anotherCar.color = "Blue";
        anotherCar.startEngine();
    }
}
```

Class vs Object

This is a fundamental distinction in OOP:

- **Class (Blueprint):**

- A template or blueprint for creating objects.
- Defines the structure (data/attributes) and behavior (methods) that all objects of that class will have.
- Does not occupy memory directly for data.
- Example: The `Dog` class defines that all dogs will have a `name`, `breed`, `age`, and can `bark()` and `displayInfo()`.

- **Object (Instance):**
  - A concrete realization or instance of a class.
  - A real-world entity that has a state (values for its attributes) and can perform behaviors.
  - Occupies memory when created using `new`.
  - Example: `myDog` and `sisterDog` are objects (instances) of the `Dog` class. `myDog` has its own `name` ("Buddy"), `breed` ("Golden Retriever"), etc., distinct from `sisterDog`'s values.

Analogy: Imagine a cookie cutter (Class). It defines the shape and pattern. When you use the cookie cutter on dough, you create actual cookies (Objects). Each cookie is distinct, even though they all came from the same cutter.

`this` & `static` Keyword

- **`this` Keyword:**
  - A reference to the *current object* (the object whose method or constructor is being called).
  - Used to:
    - Distinguish between instance variables and local variables (especially method parameters) if they have the same name.
    - Call one constructor from another constructor within the same class.
    - Pass the current object as an argument to another method.

  Example:

```java
class Box {
    int width;
    int height;

    // Constructor
    public Box(int width, int height) {
        this.width = width; // 'this.width' refers to the instance variable
        this.height = height; // 'height' refers to the parameter
    }

    // Method
    public void setDimensions(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public void printBox() {
        System.out.println("Box dimensions: " + this.width + "x" +
this.height);
    }
}

public class ThisKeywordExample {
    public static void main(String[] args) {
        Box myBox = new Box(10, 5);
        myBox.printBox(); // Output: Box dimensions: 10x5

        myBox.setDimensions(12, 6);
        myBox.printBox(); // Output: Box dimensions: 12x6
    }
}
```

- **`static` Keyword:**
  - Applies to variables, methods, and nested classes.
  - **Belongs to the class itself, not to any specific object instance.**
  - **Static Variables (Class Variables):**
    - There's only one copy of a static variable for the entire class, shared by all objects.
    - Accessed directly using the class name (`ClassName.variableName`).
  - **Static Methods (Class Methods):**
    - Can be called directly on the class (`ClassName.methodName()`) without creating an object.
    - Cannot access instance variables or call instance methods directly (because they don't belong to an object).
    - Often used for utility functions that don't depend on object state (e.g., `Math.sqrt()`).

Example:

```
class Counter {
    int instanceCount = 0;         // Instance variable: each object gets its
own
    static int staticCount = 0;    // Static variable: one copy shared by all
objects

    public Counter() {
        instanceCount++;
        staticCount++;
    }

    public void displayInstanceCount() {
        System.out.println("Instance Count for this object: " + instanceCount);
    }

    public static void displayStaticCount() {
        System.out.println("Total Static Count (shared): " + staticCount);
        // System.out.println(instanceCount); // ERROR: Cannot access
instanceCount from a static method
    }
}
```

```
public class StaticKeywordExample {
    public static void main(String[] args) {
        Counter c1 = new Counter(); // instanceCount=1, staticCount=1
        c1.displayInstanceCount(); // Output: Instance Count for this object: 1
        Counter.displayStaticCount(); // Output: Total Static Count (shared): 1

        Counter c2 = new Counter(); // instanceCount=1, staticCount=2
        c2.displayInstanceCount(); // Output: Instance Count for this object: 1
        Counter.displayStaticCount(); // Output: Total Static Count (shared): 2

        // Accessing static variable directly via class name
        System.out.println("Directly accessing staticCount: " +
Counter.staticCount);
    }
}
```

Constructors & Code Blocks

- **Constructors:**
    - Special methods used to **initialize new objects** when they are created using the `new` keyword.
    - Have the **same name as the class**.
    - Do **not have a return type** (not even `void`).
    - If you don't define any constructor, Java provides a default no-argument constructor.
    - Can be overloaded (multiple constructors with different parameters).

  Example:

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

    // No-argument constructor (default constructor if no others are defined)
    public Person() {
        this.name = "Unknown";
        this.age = 0;
        System.out.println("Person created with no arguments.");
    }

    // Parameterized constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Person created: " + name + ", " + age);
    }

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

```
public class ConstructorsExample {
    public static void main(String[] args) {
        Person p1 = new Person(); // Calls the no-argument constructor
        p1.displayInfo(); // Output: Name: Unknown, Age: 0

        Person p2 = new Person("Alice", 30); // Calls the parameterized
constructor
        p2.displayInfo(); // Output: Name: Alice, Age: 30
    }
}
```

- **Code Blocks (Initialization Blocks):**
  - Blocks of code enclosed in `{}`.
  - **Instance Initialization Blocks (`{ ... }`):**
    - Executed every time an object is created, *before* the constructor.
    - Useful for common initialization logic that applies to all constructors.
  - **Static Initialization Blocks (`static { ... }`):**
    - Executed only **once** when the class is loaded into memory (before any objects are created).
    - Used to initialize static variables or perform one-time setup tasks.

Example:

```
class Product {
    String productId;
    double price;
    static int totalProductsCreated = 0;

    // Static Initialization Block
    static {
        System.out.println("Product class loaded. Initializing static
resources.");
        // Imagine loading a configuration file or a database connection here
    }

    // Instance Initialization Block
    {
        System.out.println("A new Product object is being created.");
        this.productId = "PROD-" + (++totalProductsCreated); // Assign unique
ID
        this.price = 0.0; // Default price
    }

    // Constructor
    public Product(double price) {
        this.price = price;
        System.out.println("Product constructor called for " + productId);
    }
    public void display() {
        System.out.println("Product ID: " + productId + ", Price: $" + price);
    }
}

public class CodeBlocksExample {
    public static void main(String[] args) {
        System.out.println("--- Creating first product ---");
        Product prod1 = new Product(10.50);
        prod1.display();

        System.out.println("\n--- Creating second product ---");
        Product prod2 = new Product(25.00);
        prod2.display();

        System.out.println("\nTotal products created: " +
Product.totalProductsCreated);
    }
}
```

(Note the order of output: Static block first, then instance block, then constructor for each object.)

Stack vs Heap Memory

Java applications use two main memory areas:

- **Stack Memory:**
  - Used for **primitive type variables** and **references to objects**.
  - Stores method calls (stack frames), local variables, and return addresses.
  - Follows Last-In, First-Out (LIFO) principle.
  - Memory is allocated and deallocated very quickly.
  - When a method finishes, its stack frame is popped off, and its local variables are destroyed.
- **Heap Memory:**
  - Used for **objects** (instances of classes) and **arrays**.
  - Memory is allocated dynamically at runtime using the `new` keyword.
  - Garbage Collection primarily operates on the Heap.
  - Objects on the heap can be accessed from anywhere in the application as long as there is a reference to them.

Example Illustration (conceptual):

```
public class MemoryExample {
    public static void main(String[] args) {
        int x = 10;                    // x (primitive) stored on Stack
        String name = "Alice";         // name (reference) stored on Stack, "Alice"
(String object) stored on Heap

        Person p1 = new Person("Bob");  // p1 (reference) stored on Stack
```

```
                                        // new Person("Bob") (Person object) stored on
Heap

        modifyPerson(p1);               // When modifyPerson is called, a new stack
frame is created
                                        // inside modifyPerson, 'person' is another
reference to the SAME object on Heap
    }

    public static void modifyPerson(Person person) { // 'person' reference on Stack
        person.setName("Charlie");      // Modifies the object on Heap
    }
}

class Person { // Assume Person class with name attribute and setter
    String name;
    public Person(String name) { this.name = name; }
    public void setName(String name) { this.name = name; }
}
```

Primitive vs Reference Types

- **Primitive Types:**
    - Directly store their actual values.
    - Examples: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`.
    - Stored directly on the **Stack**.
    - When you assign one primitive to another, a *copy of the value* is made.

    ```
    int a = 10;
    int b = a; // b gets a copy of a's value (10)
    b = 20;    // Changing b does not affect a
    System.out.println("a: " + a + ", b: " + b); // Output: a: 10, b: 20
    ```

- **Reference Types:**
    - Do not store the actual object data, but rather a *reference* (memory address) to where the object is stored in the heap.
    - Examples: `String`, Arrays, any class you define (`Person`, `Car`, `Dog`), Interfaces.
    - The object data is stored on the **Heap**. The reference variable itself is stored on the Stack.
    - When you assign one reference variable to another, they both point to the *same object* on the heap.

    ```
    class MyObject {
        int value;
        public MyObject(int v) { this.value = v; }
    }

    MyObject obj1 = new MyObject(100);
    MyObject obj2 = obj1; // obj2 now refers to the SAME object as obj1
    obj2.value = 200;     // Changing obj2.value changes the object that obj1 also
    ```

```java
refers to
System.out.println("obj1.value: " + obj1.value + ", obj2.value: " +
obj2.value); // Output: obj1.value: 200, obj2.value: 200
```

Variable Scopes

The **scope** of a variable defines where in the program a variable can be accessed.

1. **Class/Static Scope (or Global Scope for static variables):**
    - `static` variables are declared inside the class, outside any method.
    - They exist for the entire lifetime of the program once the class is loaded.
    - Accessible from anywhere within the class and potentially from other classes (depending on access modifier).
2. **Instance Scope (or Object Scope):**
    - Instance variables (non-static fields) are declared inside the class, outside any method.
    - They exist as long as the object they belong to exists.
    - Accessible by all methods within that object.
3. **Method Scope (Local Variables):**
    - Variables declared inside a method.
    - They are created when the method is called and destroyed when the method finishes.
    - Only accessible within that specific method.
4. **Block Scope:**
    - Variables declared inside any code block (e.g., `if` block, `for` loop, `while` loop, arbitrary `{}` block).
    - They are created when the block is entered and destroyed when the block is exited.
    - Only accessible within that specific block.

Example:

```java
public class VariableScopesExample {
    static int classVariable = 100; // Class/Static Scope

    int instanceVariable = 200; // Instance Scope

    public void myMethod() {
        int methodVariable = 300; // Method Scope
        System.out.println("Inside myMethod:");
        System.out.println("Class Variable: " + classVariable);
        System.out.println("Instance Variable: " + instanceVariable);
        System.out.println("Method Variable: " + methodVariable);

        if (true) {
            int blockVariable = 400; // Block Scope
            System.out.println("Block Variable: " + blockVariable);
        }
        // System.out.println(blockVariable); // ERROR: blockVariable is out of scope
here
    }

    public static void main(String[] args) {
        System.out.println("Inside main method:");
        System.out.println("Class Variable: " + classVariable); // Accessible

        VariableScopesExample obj = new VariableScopesExample();
        System.out.println("Instance Variable via object: " + obj.instanceVariable);
// Accessible via object

        obj.myMethod(); // Calls the method, which accesses its variables
        // System.out.println(methodVariable); // ERROR: methodVariable is out of
scope here
    }
}
```

Garbage Collection & Finalize

- **Garbage Collection (GC):**
    - Java has an automatic memory management system. You don't explicitly deallocate memory for objects you create.
    - The **Garbage Collector** is a daemon thread that runs in the background. Its job is to identify and reclaim memory occupied by objects that are no longer "reachable" (i.e., no longer referenced by any active part of the program).
    - This prevents memory leaks and simplifies memory management for the developer.
    - You can *suggest* GC to run using `System.gc()`, but there's no guarantee when or if it will execute.
- **Finalize Method (`protected void finalize() throws Throwable`):**
    - This method (part of the `Object` class) is called by the Garbage Collector *just before* an object is actually destroyed.
    - It's meant for cleanup operations, like closing database connections or releasing system resources *if they are not handled by other means* (like `try-with-resources` for files/connections, which is highly preferred).
    - **Important Note:** `finalize()` is generally **discouraged** in modern Java programming due to several issues:
        - No guarantee it will ever run (GC might not run for short-lived programs).
        - No guarantee *when* it will run.
        - Can delay garbage collection.
        - Exceptions thrown in `finalize()` are ignored.
        - Modern resource management (e.g., `try-with-resources`) is far more reliable and preferred.

Example (demonstrating `finalize` conceptually, but rarely used in practice):

```
class ResourceUser {
    String name;
```

```java
    public ResourceUser(String name) {
        this.name = name;
        System.out.println(name + " created.");
    }

    // This method is called by the Garbage Collector before destroying the object
    @Override
    protected void finalize() throws Throwable {
        System.out.println(name + " is being finalized (memory reclaimed).");
        // Imagine closing a file handle or network connection here
        super.finalize(); // Call superclass finalize if needed
    }
}

public class GarbageCollectionExample {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("Creating some objects...");
        createObjects(); // Objects created here will become eligible for GC after
method finishes

        System.out.println("\nSuggesting Garbage Collection...");
        // This is just a suggestion, GC might not run immediately
        System.gc();

        // Give the GC some time to potentially run and finalize objects
        Thread.sleep(1000); // Wait for 1 second

        System.out.println("\nProgram finished.");
    }

    public static void createObjects() {
        ResourceUser obj1 = new ResourceUser("Object A");
        ResourceUser obj2 = new ResourceUser("Object B");
        // obj1 and obj2 become unreachable after this method exits
    }
}
```

Control Statements, Math & String

Ternary Operator

The **ternary operator** (also known as the conditional operator) is a shorthand way to write a simple `if-else` statement. It's the only operator in Java that takes three operands.

Syntax: `condition ? expressionIfTrue : expressionIfFalse;`

How it works:

- `condition` is evaluated.
- If `condition` is `true`, `expressionIfTrue` is evaluated and its result is returned.

- If `condition` is `false`, `expressionIfFalse` is evaluated and its result is returned.

Example:

```java
public class TernaryOperatorExample {
    public static void main(String[] args) {
        int age = 20;
        String eligibility;

        // Using if-else
        if (age >= 18) {
            eligibility = "Eligible to vote";
        } else {
            eligibility = "Not eligible to vote";
        }
        System.out.println("Using if-else: " + eligibility);

        // Using ternary operator (more concise for simple conditions)
        eligibility = (age >= 18) ? "Eligible to vote" : "Not eligible to vote";
        System.out.println("Using ternary: " + eligibility);

        int num = 7;
        String type = (num % 2 == 0) ? "Even" : "Odd";
        System.out.println(num + " is " + type); // Output: 7 is Odd

        int value1 = 10, value2 = 20;
        int max = (value1 > value2) ? value1 : value2;
        System.out.println("Max of " + value1 + " and " + value2 + " is: " + max); //
Output: 20
    }
}
```

Switch

The `switch` **statement** is a control flow statement that allows a variable to be tested for equality against a list of values. It provides a more elegant way to handle multiple `if-else if` conditions when comparing a single variable against several discrete values.

Syntax:

```java
switch (expression) {
    case value1:
        // code block
        break; // Optional, but usually used to exit the switch
    case value2:
        // code block
        break;
    // ...
    default: // Optional: executed if no case matches
        // code block
}
```

- `expression`: Can be `byte`, `short`, `int`, `char`, `String`, `enum`, or wrapper classes for these types (from Java 7 onwards for `String`).
- `break`: Exits the `switch` block. Without `break`, execution will "fall through" to the next `case`.
- `default`: An optional block executed if no `case` matches the `expression`.

Example:

```java
public class SwitchExample {
    public static void main(String[] args) {
        int dayOfWeek = 3; // 1=Monday, 2=Tuesday, etc.
        String dayName;

        switch (dayOfWeek) {
            case 1:
                dayName = "Monday";
                break;
            case 2:
                dayName = "Tuesday";
                break;
            case 3:
                dayName = "Wednesday";
                break;
            case 4:
                dayName = "Thursday";
                break;
            case 5:
                dayName = "Friday";
                break;
            case 6:
            case 7: // Multiple cases can share the same code block
                dayName = "Weekend";
                break;
            default:
                dayName = "Invalid Day";
                break;
        }
        System.out.println("Day " + dayOfWeek + " is: " + dayName); // Output: Day 3
is: Wednesday

        char grade = 'B';
        switch (grade) {
            case 'A':
                System.out.println("Excellent!");
                break;
            case 'B':
                System.out.println("Very Good!");
                break;
            case 'C':
                System.out.println("Good!");
                break;
            default:
                System.out.println("Needs improvement.");
        }

        // Java 14+ introduced 'switch expressions' with 'yield'
        // String season = switch (month) { ... };
        // This is a more modern way, but the traditional switch statement is still
common.
    }
}
```

Loops (Do-while, For, For-each)

We've already covered the `while` loop. Here are the other common loop types in Java:

1. **`do-while` Loop:**

   - Similar to `while`, but guarantees that the loop body is executed at least once, because the condition is checked *after* the first iteration.

   Syntax:

   ```
   do {
       // Code to be executed
   } while (condition);
   ```

   Example:

   ```
   public class DoWhileLoopExample {
       public static void main(String[] args) {
           int count = 1;
           System.out.println("Counting up (do-while):");
           do {
               System.out.println("Count: " + count);
               count++;
           } while (count <= 5); // Condition check happens after printing 1st
   time

           int input;
           java.util.Scanner scanner = new java.util.Scanner(System.in);
           do {
               System.out.print("Enter a number between 1 and 10: ");
               input = scanner.nextInt();
           } while (input < 1 || input > 10);
           System.out.println("You entered a valid number: " + input);
           scanner.close();
       }
   }
   ```

2. **`for` Loop:**

   - Used when you know exactly how many times you want to iterate, or when you need a clear initialization, condition, and update step.

   Syntax:

```
for (initialization; condition; update) {
    // Code to be executed
}
```

Example:

```
public class ForLoopExample {
    public static void main(String[] args) {
        // Counting from 0 to 4
        System.out.println("Counting with for loop:");
        for (int i = 0; i < 5; i++) {
            System.out.println("Iteration: " + i);
        }

        // Counting down
        System.out.println("\nCounting down:");
        for (int i = 5; i >= 1; i--) {
            System.out.println("Countdown: " + i);
        }

        // Summing numbers
        int sum = 0;
        for (int i = 1; i <= 10; i++) {
            sum += i;
        }
        System.out.println("\nSum of 1 to 10: " + sum); // Output: 55
    }
}
```

3. **`for-each` Loop (Enhanced for Loop):**
    - Designed for iterating over collections (like arrays) where you don't need the index. It's more concise and less prone to off-by-one errors.

Syntax:

```
for (dataType elementVariable : collection) {
    // Code to be executed for each element
}
```

Example:

```
public class ForEachLoopExample {
    public static void main(String[] args) {
        String[] fruits = {"Apple", "Banana", "Cherry"};

        System.out.println("Fruits (using for-each):");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }

        int[] scores = {85, 92, 78, 95};
        int totalScore = 0;
        for (int score : scores) {
            totalScore += score;
        }
        System.out.println("\nTotal score: " + totalScore); // Output: 350
    }
}
```

Using `break` & `continue`

- **`break` statement:**
    - Immediately terminates the innermost loop (or `switch` statement) it is in.
    - Execution continues with the statement immediately following the loop/switch.
- **`continue` statement:**
    - Skips the rest of the current iteration of the innermost loop.
    - The loop proceeds to the next iteration (evaluating the condition for `while` / `do-while`, or the update part for `for`).

Example:

```
public class BreakContinueExample {
    public static void main(String[] args) {
        System.out.println("--- Using break ---");
        for (int i = 0; i < 10; i++) {
            if (i == 5) {
                System.out.println("Breaking loop at i = " + i);
                break; // Exit the loop entirely
            }
            System.out.println("Current i: " + i);
        }
        System.out.println("Loop finished (after break).");

        System.out.println("\n--- Using continue ---");
        for (int i = 0; i < 5; i++) {
            if (i == 2) {
                System.out.println("Skipping iteration at i = " + i);
                continue; // Skip the rest of this iteration, go to next i
            }
            System.out.println("Current i: " + i);
        }
        System.out.println("Loop finished (after continue).");

        // Example with nested loops and labels (less common, but possible)
        outerLoop:
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (i == 1 && j == 1) {
                    System.out.println("Breaking outer loop at i=" + i + ", j=" + j);
                    break outerLoop; // Breaks out of the labeled outer loop
                }
                System.out.println("i: " + i + ", j: " + j);
            }
        }
    }
}
```

Recursion

Recursion is a programming technique where a method calls itself to solve a problem. A recursive solution typically involves:

1. **Base Case:** A condition that stops the recursion (prevents infinite calls).
2. **Recursive Step:** The method calls itself with a smaller version of the problem, moving closer to the base case.

Example: Factorial Calculation `n! = n * (n-1)!` `Base case: 0! = 1`

```
public class RecursionExample {

    // Method to calculate factorial using recursion
    public static int factorial(int n) {
        // Base case: if n is 0, return 1
        if (n == 0) {
            return 1;
        }
        // Recursive step: n * factorial(n-1)
        return n * factorial(n - 1);
    }

    // Another example: Fibonacci sequence
    // Fib(0) = 0, Fib(1) = 1, Fib(n) = Fib(n-1) + Fib(n-2)
    public static int fibonacci(int n) {
        if (n <= 1) { // Base cases
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2); // Recursive step
    }

    public static void main(String[] args) {
        int num = 5;
        System.out.println("Factorial of " + num + ": " + factorial(num)); // Output:
120

        int fibNum = 7;
        System.out.println("Fibonacci of " + fibNum + ": " + fibonacci(fibNum)); //
Output: 13
    }
}
```

Random Numbers & Math Class

- **`Math` Class:**

    - A utility class in `java.lang` (so no import needed) that provides static methods for common mathematical functions.
    - Examples: `Math.abs()`, `Math.sqrt()`, `Math.pow()`, `Math.round()`, `Math.ceil()`, `Math.floor()`, `Math.max()`, `Math.min()`, `Math.random()`.
    - `Math.random()`: Returns a `double` value between `0.0` (inclusive) and `1.0` (exclusive).
- **`Random` Class (`java.util.Random`):**
    - Provides more robust methods for generating various types of random numbers.
    - You create an instance of `Random` and then call methods like `nextInt()`, `nextBoolean()`, `nextDouble()`.

Example:

```java
import java.util.Random;

public class RandomAndMathExample {
    public static void main(String[] args) {
        // --- Using Math Class ---
        System.out.println("--- Math Class Examples ---");
        System.out.println("Absolute value of -10: " + Math.abs(-10)); // 10
        System.out.println("Square root of 25: " + Math.sqrt(25));   // 5.0
        System.out.println("2 raised to the power of 3: " + Math.pow(2, 3)); // 8.0
        System.out.println("Round 4.7: " + Math.round(4.7));         // 5
        System.out.println("Ceil 4.2: " + Math.ceil(4.2));           // 5.0 (smallest integer >= value)
        System.out.println("Floor 4.7: " + Math.floor(4.7));         // 4.0 (largest integer <= value)
        System.out.println("Max of 10 and 20: " + Math.max(10, 20)); // 20
        System.out.println("Min of 10 and 20: " + Math.min(10, 20)); // 10

        System.out.println("\nRandom double from Math.random(): " + Math.random()); // 0.0 to < 1.0

        // Generating a random integer between 1 and 10 using Math.random()
        int randomInt = (int) (Math.random() * 10) + 1; // (0.0 to 0.999...) * 10 -> (0.0 to 9.999...) + 1 -> (1.0 to 10.999...) cast to int
        System.out.println("Random int (1-10) using Math.random(): " + randomInt);

        // --- Using Random Class ---
        System.out.println("\n--- Random Class Examples ---");
        Random rand = new Random(); // Create a Random object

        System.out.println("Next random int: " + rand.nextInt()); // Full range of int
        System.out.println("Next random int (0-99): " + rand.nextInt(100)); // 0 (inclusive) to 100 (exclusive)
        System.out.println("Next random boolean: " + rand.nextBoolean());
        System.out.println("Next random double: " + rand.nextDouble()); // 0.0 (inclusive) to 1.0 (exclusive)

        // Generating a random integer between 1 and 6 (for a dice roll)
        int diceRoll = rand.nextInt(6) + 1; // rand.nextInt(6) gives 0-5, so add 1 for
1-6
        System.out.println("Dice roll: " + diceRoll);
    }
}
```

Don’t Learn Syntax (Conceptual)

This point is more of a philosophy than a specific Java topic. The idea is that while knowing syntax is necessary, **true programming skill lies in understanding the underlying concepts, logic, and problem-solving techniques**, rather than just memorizing grammar rules.

- **Focus on concepts:** Understand *why* you use a `for` loop versus a `while` loop, *when* to use a class versus just functions, *how* recursion solves problems, etc.
- **Practice:** The best way to internalize syntax is by consistently writing code. The syntax will become second nature through practice.
- **Don't fear errors:** Syntax errors are common. They help you learn the rules. Use your IDE (Integrated Development Environment) to catch them and learn from the feedback.
- **Refer to documentation:** When you forget a specific syntax detail, look it up! No one memorizes everything. The ability to find and understand documentation is a crucial skill.

`toString()` Method

The `toString()` method is a special method available in every Java class (because all classes implicitly inherit from `Object`, which defines `toString()`).

- Its purpose is to provide a **string representation of an object**.
- By default, `Object`'s `toString()` returns a string like `ClassName@hashCode`. This is usually not very helpful.
- It's a common practice to **override** the `toString()` method in your own classes to provide a meaningful and descriptive representation of your objects' state. This is especially useful for debugging and logging.

Example:

```
class Book {
    String title;
    String author;
    int year;

    public Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.year = year;
    }

    // Overriding the toString() method
    @Override
    public String toString() {
        return "Book [Title: " + title + ", Author: " + author + ", Year: " + year +
"]";
    }
}

public class ToStringExample {
    public static void main(String[] args) {
        Book book1 = new Book("The Lord of the Rings", "J.R.R. Tolkien", 1954);
        Book book2 = new Book("Pride and Prejudice", "Jane Austen", 1813);

        // When you print an object, Java implicitly calls its toString() method
        System.out.println(book1); // Output: Book [Title: The Lord of the Rings,
Author: J.R.R. Tolkien, Year: 1954]
        System.out.println(book2.toString()); // Explicitly calling toString() works
the same

        // If toString() wasn't overridden, output would look like: Book@1b6d3586
    }
}
```

String class

The `String` class in Java (`java.lang.String`) is used to store and manipulate sequences of characters (text).

Key characteristics:

- **Immutable:** 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.
- `String` literals are stored in the String Pool (a special area in the heap) for efficiency.
- Widely used and provides a rich set of methods for operations like concatenation, comparison, searching, substring extraction, case conversion, etc.

Example:

```
public class StringClassExample {
    public static void main(String[] args) {
        // String Literals
        String s1 = "Hello";
        String s2 = "World";

        // Creating String objects using new keyword
        String s3 = new String("Java");
        String s4 = new String("Java");

        // Concatenation
        String greeting = s1 + " " + s2 + "!";
        System.out.println("Concatenation: " + greeting); // Hello World!
        String welcome = s1.concat(" ").concat(s3);
        System.out.println("Concatenation with concat(): " + welcome); // Hello Java

        // Length
        System.out.println("Length of greeting: " + greeting.length()); // 12
        // Accessing characters
        System.out.println("Character at index 0 in s1: " + s1.charAt(0)); // H

        // Substring
        System.out.println("Substring of greeting (index 6 to end): " + greeting.substring(6)); // World!
        System.out.println("Substring of greeting (index 0 to 5): " + greeting.substring(0, 5)); // Hello

        // Comparison
        System.out.println("s1 equals s2: " + s1.equals(s2)); // false
        System.out.println("s3 equals s4: " + s3.equals(s4)); // true (compares content)
        System.out.println("s3 == s4: " + (s3 == s4));        // false (compares references, they are different objects)
        String s5 = "Java"; // This will likely refer to the same "Java" in the String Pool as s3
        System.out.println("s3 == s5: " + (s3 == s5));        // false (still different objects if s3 was new String)
        System.out.println("s5 == \"Java\": " + (s5 == "Java")); // true (both refer to String Pool literal)

        System.out.println("s1 equalsIgnoreCase \"hello\": " + s1.equalsIgnoreCase("hello")); // true

        // Case conversion
        System.out.println("greeting to lowercase: " + greeting.toLowerCase());
        System.out.println("greeting to uppercase: " + greeting.toUpperCase());

        // Searching
        System.out.println("Index of 'o' in s1: " + s1.indexOf('o')); // 4
        System.out.println("Contains \"Wor\": " + greeting.contains("Wor")); // true

        // Replacement
        String modified = greeting.replace("World", "Universe");
        System.out.println("Replaced string: " + modified); // Hello Universe!

        // Trimming whitespace
        String padded = "   Trim me   ";
        System.out.println("Padded: '" + padded + "'");
        System.out.println("Trimmed: '" + padded.trim() + "'");
    }
}
```

`StringBuffer` vs `StringBuilder`

Both `StringBuffer` and `StringBuilder` are mutable sequence of characters. This means their content *can* be changed after creation, unlike `String`. They are used when you need to perform many modifications to a string (e.g., in a loop), as creating many immutable `String` objects can be inefficient.

- **`StringBuffer`:**

    - **Synchronized (thread-safe):** Its methods are synchronized, meaning only one thread can access them at a time. This makes it safe for use in multi-threaded environments, but incurs a performance overhead.
    - **Slower** than `StringBuilder`.
- **`StringBuilder`:**
    - **Non-synchronized (not thread-safe):** Its methods are not synchronized.
    - **Faster** than `StringBuffer` because it doesn't have the synchronization overhead.
    - **Preferred** for single-threaded environments (which is most common).

Example:

```
public class StringBufferStringBuilderExample {
    public static void main(String[] args) {
        System.out.println("--- Using StringBuilder (preferred for single-threaded) --
-");
        StringBuilder sb = new StringBuilder("Initial");
        sb.append(" text"); // Appends to the same object
        sb.insert(7, " new"); // Inserts at index 7
        sb.replace(0, 7, "Changed"); // Replaces a portion
        sb.delete(13, 17); // Deletes a portion
        System.out.println(sb); // Output: Changed newtext

        sb.reverse(); // Reverses the string
        System.out.println("Reversed: " + sb); // txetwen degnahC

        // Convert back to String when done
        String finalString = sb.toString();
        System.out.println("Final String: " + finalString);


        System.out.println("\n--- Using StringBuffer (for multi-threaded scenarios) --
-");
        StringBuffer sbuf = new StringBuffer("Start");
        sbuf.append(" working");
        sbuf.delete(5, 7); // Delete " w"
        sbuf.insert(5, " and learning");
        System.out.println(sbuf); // Output: Start and learningorking

        // Also convertible to String
        String finalBufferString = sbuf.toString();
        System.out.println("Final Buffer String: " + finalBufferString);
    }
}
```

`final` Keyword

The `final` keyword in Java is used to restrict the user. It can be applied to:

1. **`final` Variable:**

- Makes a variable a **constant**. Its value can be assigned only once (either at declaration or in a constructor/initializer block).
- For primitive types, the value cannot change.
- For reference types, the *reference* cannot be changed (i.e., it cannot point to a different object), but the contents of the object it points to *can* be modified (unless the object itself is immutable, like `String`).
- Commonly used with `static` to create **class-level constants** (e.g., `public static final double PI = 3.14159;`).

Example:

```
public class FinalVariableExample {
    final int MAX_SPEED = 120; // Final instance variable, must be initialized
    // MAX_SPEED = 130; // ERROR: cannot assign a value to final variable

    final String APP_NAME; // Can be initialized in constructor

    public FinalVariableExample() {
        APP_NAME = "My Application"; // Initialized here
    }

    public void printConstants() {
        System.out.println("Max Speed: " + MAX_SPEED);
        System.out.println("App Name: " + APP_NAME);
    }

    public static void main(String[] args) {
        final int COUNT = 10; // Final local variable
        // COUNT = 11; // ERROR: cannot assign a value to final variable

        final StringBuilder sb = new StringBuilder("Hello"); // sb reference is
final
        sb.append(" World"); // OK: object content can be modified
        // sb = new StringBuilder("Goodbye"); // ERROR: cannot reassign final
reference

        FinalVariableExample example = new FinalVariableExample();
        example.printConstants();
    }
}
```

2. **`final` Method:**

- Prevents a method from being **overridden** by subclasses.
- Ensures that the behavior of that method remains consistent across the class hierarchy.

Example:

```
class Parent {
    public final void importantMethod() {
        System.out.println("This is an important method that cannot be
overridden.");
```

```java
    }

    public void normalMethod() {
        System.out.println("This is a normal method.");
    }
}

class Child extends Parent {
    // @Override
    // public void importantMethod() { // ERROR: cannot override final method
    //     System.out.println("Trying to override...");
    // }

    @Override
    public void normalMethod() {
        System.out.println("Child's version of normal method.");
    }
}
```

3. **`final` Class:**
   - Prevents a class from being **inherited** (no other class can extend it).
   - Used for security or design reasons, for example, `String` class is `final` to ensure immutability and security.

   Example:

```java
final class ImmutableClass {
    private final int value;
    public ImmutableClass(int value) {
        this.value = value;
    }
    public int getValue() { return value; }
}

// class MySubClass extends ImmutableClass { // ERROR: cannot inherit from final class
// }
```

Encapsulation & Inheritance

Intro to OOPs Principles

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects". It is designed to make code more modular, flexible, and reusable. There are four core principles:

1. **Encapsulation:** The bundling of data (attributes) and the methods that operate on that data into a single unit called a class. It's about data hiding and protecting data from outside interference.
2. **Inheritance:** The mechanism by which one class (subclass or child) acquires the properties and behaviors (methods and fields) of another class (superclass or parent). It promotes code reuse and establishes a relationship between classes.

3. **Polymorphism:** The ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object, allowing the same method name to behave differently for different objects.
4. **Abstraction:** The concept of hiding the complex implementation details and showing only the essential features of the object. It helps in managing complexity. An interface is a common way to achieve abstraction.

What is Encapsulation

Encapsulation is the practice of bundling an object's data (instance variables) and the methods that operate on that data into a single unit (a class). A key part of encapsulation is **data hiding**, which is achieved by making the instance variables `private` so they cannot be accessed directly from outside the class. Access to this data is then controlled through public methods, typically **getters** and **setters**.

Benefits of Encapsulation:

- **Control:** You have full control over the data. You can add validation logic in setter methods to ensure the data remains in a valid state.
- **Security:** Data hiding protects the internal state of an object from accidental or unauthorized modification.
- **Flexibility & Maintainability:** The internal implementation of the class can be changed without affecting the code that uses the class, as long as the public methods remain the same.
- **Read-Only or Write-Only Classes:** You can create a read-only class by providing only a getter method, or a write-only class with only a setter.

Example:

```java
public class BankAccount {
    // 1. Private instance variable (data hiding)
    private double balance;

    public BankAccount(double initialBalance) {
        if (initialBalance > 0) {
            this.balance = initialBalance;
        } else {
            this.balance = 0;
        }
    }

    // 2. Public method to deposit money (controlled access)
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: $" + amount);
        } else {
            System.out.println("Cannot deposit a negative amount.");
        }
    }

    // 3. Public method to withdraw money (controlled access with validation)
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrew: $" + amount);
        } else {
            System.out.println("Withdrawal failed. Invalid amount or insufficient
funds.");
        }
    }

    // 4. Public method to check the balance (controlled access)
    public double getBalance() {
        return balance;
    }
}

// Another class trying to use BankAccount
public class BankClient {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(100.0);

        // account.balance = -500; // ERROR! Cannot access private data directly. This
is encapsulation in action.

        System.out.println("Initial balance: $" + account.getBalance());

        account.deposit(50.0);
        account.withdraw(30.0);
        account.withdraw(200.0); // Fails due to insufficient funds

        System.out.println("Final balance: $" + account.getBalance());
    }
}
```

Import & Packages

- **Packages:** A package in Java is a way to group related classes and interfaces. They serve as namespaces to prevent naming conflicts. For example, you can have a `com.company.math.Calculator` class and a `com.othercompany.ui.Calculator` class without them clashing. Packages correspond to the directory structure of your project. The convention is to use a reversed domain name (e.g., `com.google.common`).
- **`import` Statement:** The `import` statement allows you to refer to classes and interfaces from other packages by their simple names instead of their fully qualified names.

Example: Without `import`, you would have to write: `java.util.Scanner scanner = new java.util.Scanner(System.in);`

With `import`, it becomes much cleaner: `import java.util.Scanner;` `Scanner scanner = new Scanner(System.in);`

You can also import all classes from a package using the wildcard `*`: `import java.util.*; // Imports all classes in the java.util package`

Access Modifiers

Access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class.

1. **`public`:** The member is accessible from anywhere (from other classes, other packages, etc.). This is the least restrictive modifier.
2. **`protected`:** The member is accessible within its own package and by subclasses (even if they are in different packages).
3. **Default (no modifier):** If you don't specify any modifier, the member is accessible only within its own package. It's also called "package-private".
4. **`private`:** The member is accessible only within its own class. This is the most restrictive modifier and is fundamental to encapsulation.

Modifier	Same Class	Same Package	Subclass (diff. package)	World (diff. package)
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
`private`	Yes	No	No	No

Getter and Setter

Getters and setters are public methods used to protect your data, especially in the context of encapsulation.

- **Getter Method:** A method that reads the value of a `private` instance variable. By convention, its name starts with `get` followed by the variable name (e.g., `getBalance()`). For boolean variables, it may start with `is` (e.g., `isEnabled()`).
- **Setter Method:** A method that modifies or updates the value of a `private` instance variable. By convention, its name starts with `set` followed by the variable name (e.g., `setBalance(double amount)`).

Example (revisiting the `BankAccount`):

```java
public class Employee {
    private String name;
    private int employeeId;

    // Getter for 'name'
    public String getName() {
        return name;
    }

    // Setter for 'name'
    public void setName(String name) {
        if (name != null && !name.trim().isEmpty()) { // Validation logic
            this.name = name;
        }
    }

    // Getter for 'employeeId' (read-only)
    public int getEmployeeId() {
        return employeeId;
    }

    // No setter for employeeId, making it effectively a read-only property after
object creation.
    public Employee(int employeeId, String name) {
        this.employeeId = employeeId;
        this.setName(name); // Use the setter to apply validation
    }
}
```

What is Inheritance

Inheritance is a core OOP concept where a new class (the **subclass** or **child class**) is based on an existing class (the **superclass** or **parent class**). The child class inherits the public and protected attributes and methods of the parent class.

Key benefits:

- **Code Reusability:** You can write common code once in a parent class and reuse it across multiple child classes.
- **Method Overriding:** Child classes can provide their own specific implementation of a method that is already defined in the parent class.
- **Is-A Relationship:** Inheritance establishes a logical "is-a" relationship. For example, a `Dog` is an `Animal`. This is crucial for polymorphism.

The `extends` keyword is used for inheritance.

Example:

```
// Parent class (Superclass)
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }

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

// Child class (Subclass) inherits from Animal
class Dog extends Animal {
```

```java
    public Dog(String name) {
        super(name); // 'super' calls the constructor of the parent class
    }

    // Method Overriding: Providing a specific implementation for a parent method
    @Override
    public void makeSound() {
        System.out.println("Woof! Woof!");
    }

    // New method specific to Dog
    public void fetch() {
        System.out.println(name + " is fetching the ball.");
    }
}

public class InheritanceExample {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy");
        myDog.eat(); // Inherited from Animal
        myDog.makeSound(); // Overridden method in Dog is called
        myDog.fetch(); // Method specific to Dog
    }
}
```

Types of Inheritance

Java supports several types of inheritance through classes and interfaces:

1. **Single Inheritance:** A class can inherit from only one superclass. (e.g., `Dog extends Animal`). This is the only form of class inheritance supported by Java to avoid complexity.
2. **Multilevel Inheritance:** A class inherits from a child class, forming a chain. (e.g., `class Puppy extends Dog`, where `Dog extends Animal`).
3. **Hierarchical Inheritance:** Multiple classes inherit from a single superclass. (e.g., `Dog extends Animal` and `Cat extends Animal`).

Not Supported in Java (for classes):

- **Multiple Inheritance:** A class cannot inherit from more than one class. This is disallowed to prevent the "Diamond Problem," where ambiguity arises if two parent classes have a method with the same signature. Java solves this by allowing a class to *implement* multiple *interfaces*.
- **Hybrid Inheritance:** A combination of multiple and multilevel inheritance. Also not supported for classes.

Object Class

The `java.lang.Object` class is the root of the class hierarchy in Java. **Every class in Java is a direct or indirect subclass of `Object`**. If you create a class that does not explicitly `extend` another class, it implicitly `extends Object`.

This means that every object in Java inherits the methods of the `Object` class. Some of the most important methods are:

- `toString()` : Returns a string representation of the object.
- `equals(Object obj)` : Compares two objects for equality.
- `hashCode()` : Returns a hash code (an integer) for the object, used in hash-based collections like `HashMap` .
- `getClass()` : Returns the runtime class of the object.
- `finalize()` : Called by the garbage collector before an object is reclaimed. (Discouraged)
- `clone()` : Creates a copy of the object.

`equals()` and `hashCode()`

These two methods from the `Object` class are crucial when working with collections.

- **`equals(Object obj)` :**
    - The default implementation in the `Object` class checks for reference equality (`==`), meaning it returns `true` only if two references point to the exact same object in memory.
    - You should **override `equals()`** to define a custom "logical equality" for your objects. For example, two `Employee` objects might be considered equal if they have the same `employeeId` .
- **`hashCode()` :**
    - Returns an integer hash code for the object. This code is used by data structures like `HashMap` , `HashSet` , and `Hashtable` to store and retrieve objects efficiently.

The `equals()` and `hashCode()` Contract: There is a strict contract between these two methods:

1. **If `obj1.equals(obj2)` is true, then `obj1.hashCode()` must be equal to `obj2.hashCode()`** .
2. If `obj1.equals(obj2)` is false, their hash codes do *not* have to be different, but for performance, it's highly desirable that they are.

Rule of Thumb: Whenever you override `equals()` , you MUST also override `hashCode()` . If you fail to do this, your objects will behave incorrectly when stored in hash-based collections.

Example:

```java
import java.util.Objects;

class Student {
    private int studentId;
    private String name;

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

    // Overriding equals()
    @Override
    public boolean equals(Object o) {
        // 1. Check if the object is being compared with itself
        if (this == o) return true;
        // 2. Check if the other object is null or of a different class
        if (o == null || getClass() != o.getClass()) return false;
        // 3. Cast the object to the correct type
        Student student = (Student) o;
        // 4. Compare the relevant fields for equality
        return studentId == student.studentId;
    }

    // Overriding hashCode()
    @Override
    public int hashCode() {
        // Use the same fields that were used in the equals() method
        return Objects.hash(studentId);
    }
}

public class EqualsHashCodeExample {
    public static void main(String[] args) {
        Student s1 = new Student(101, "Alice");
        Student s2 = new Student(101, "Alice V2"); // Same ID, different name
        Student s3 = new Student(102, "Bob");

        System.out.println("s1 equals s2: " + s1.equals(s2)); // true (because
studentId is the same)
        System.out.println("s1 equals s3: " + s1.equals(s3)); // false

        System.out.println("s1 hash code: " + s1.hashCode());
        System.out.println("s2 hash code: " + s2.hashCode()); // Must be the same as
s1's
        System.out.println("s3 hash code: " + s3.hashCode()); // Likely different
    }
}
```

Nested and Inner Classes

Java allows you to define a class within another class. Such a class is called a **nested class**. They are used to group classes that are only used in one place, which increases encapsulation and makes the code more readable and maintainable.

There are two main types of nested classes:

1. **Static Nested Class:**

    - Declared with the `static` keyword.
    - Behaves like a regular top-level class but is enclosed within the namespace of the outer class.
    - **Cannot** access the instance members (non-static fields and methods) of the outer class directly. It can only access static members of the outer class.
    - Instantiated using the outer class name: `OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();`

2. **Inner Class (Non-static Nested Class):**
    - Is associated with an *instance* of the outer class.
    - **Can** access all members (both static and non-static) of the outer class, including `private` ones.
    - An instance of an inner class can only exist within an instance of the outer class.
    - Instantiated using an instance of the outer class: `OuterClass.InnerClass innerObject = outerObject.new InnerClass();`

There are also two special kinds of inner classes:

- **Local Inner Class:** A class defined inside a method. It is only visible within that method.
- **Anonymous Inner Class:** A class without a name, used for one-time use, often for implementing an interface or extending a class on the fly (common in event handling).

Example:

```
class OuterClass {
    private String outerField = "Outer field";
    private static String staticOuterField = "Static outer field";

    // Static Nested Class
    static class StaticNestedClass {
        void display() {
            System.out.println("Static Nested Class accessing: " + staticOuterField);
            // System.out.println(outerField); // ERROR: Cannot access non-static
member
        }
    }

    // Inner Class (Non-static)
    class InnerClass {
        void display() {
            // Can access both static and non-static members of the outer class
            System.out.println("Inner Class accessing: " + outerField);
            System.out.println("Inner Class also accessing: " + staticOuterField);
        }
    }

    public void createAndShowInner() {
        // Example of creating a Local Inner Class
        class LocalInner {
            void show() {
                System.out.println("Local Inner Class says: Hello!");
            }
        }
        LocalInner li = new LocalInner();
        li.show();
    }
}
```

```
public class NestedClassExample {
    public static void main(String[] args) {
        // Instantiating a static nested class
        OuterClass.StaticNestedClass staticNested = new
OuterClass.StaticNestedClass();
        staticNested.display();

        // Instantiating an inner class requires an instance of the outer class
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display();

        // Calling the method that uses a local inner class
        outer.createAndShowInner();
    }
}
```

Abstraction and Polymorphism

What is Abstraction

Abstraction is the OOP principle of hiding the complex implementation details from the user and showing only the essential features or functionality. It helps manage complexity by focusing on the "what" an object does, rather than the "how" it does it.

Think about driving a car. To accelerate, you press the gas pedal. You don't need to know about the engine's combustion process, the fuel injection system, or the transmission's gear shifts. The complex internal workings are hidden (abstracted away), and you are provided with a simple interface (the pedal) to achieve your goal.

In Java, abstraction is achieved using **abstract classes** and **interfaces**. The goal is to create a contract that implementing or extending classes must follow.

Key Benefits of Abstraction:

- **Simplicity:** Hides complexity from the user.
- **Reduces Impact of Change:** The underlying implementation can be changed without affecting the classes that depend on it, as long as the abstract contract is maintained.
- **Manages Complexity:** Breaks down complex systems into smaller, more manageable, and understandable components.

Abstract Keyword

The `abstract` keyword is a non-access modifier in Java used to achieve abstraction for classes and methods.

1. **Abstract Class:**
    - An `abstract class` is a class that is declared with the `abstract` keyword.
    - It **cannot be instantiated** (you cannot create an object of an abstract class using `new`). Its purpose is to be extended by other classes.
    - It can contain both **abstract methods** (methods without a body) and **concrete methods** (regular methods with a body).

    - It can have constructors, static methods, and `final` methods. The constructor is called when an instance of a concrete subclass is created.
    - Any class that extends an abstract class must either implement all of the parent's abstract methods or be declared abstract itself.

2. **Abstract Method:**

    - An `abstract method` is a method that is declared with the `abstract` keyword and has no implementation (no body, just a signature followed by a semicolon).
    - It can only be declared inside an `abstract class` or an `interface`.
    - It acts as a contract, forcing any non-abstract subclass to provide a specific implementation for that method.

Example:

Let's model different shapes. All shapes have an area, but the formula to calculate the area is different for each one. This is a perfect scenario for abstraction.

```java
// 1. Abstract Class: Defines the "idea" or "contract" of a Shape.
// You can't create a generic "Shape" object, it must be something specific like a Circle or Rectangle.
abstract class Shape {
    String color;

    // Abstract method: has no body.
    // It declares that every subclass of Shape MUST provide an implementation for calculating its area.
    public abstract double calculateArea();

    // Concrete method: has a body and is inherited by all subclasses.
    public void display() {
        System.out.println("Displaying a shape.");
    }

    // Abstract classes can have constructors.
    public Shape(String color) {
        this.color = color;
        System.out.println("Shape constructor called.");
    }

    public String getColor() {
        return color;
    }
}

// 2. Concrete Subclass: Implements the contract defined by the abstract class.
class Circle extends Shape {
    double radius;

    public Circle(String color, double radius) {
        // Call the superclass (Shape) constructor
        super(color);
        this.radius = radius;
        System.out.println("Circle constructor called.");
    }

    // Providing the mandatory implementation for the abstract method.
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// 3. 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;
        System.out.println("Rectangle constructor called.");
    }

    // Providing its own specific implementation for the abstract method.
    @Override
    public double calculateArea() {
        return width * height;
    }
}

public class AbstractionExample {
    public static void main(String[] args) {
        // Shape myShape = new Shape("Red"); // ERROR! Cannot instantiate an abstract
class.

        // Create objects of concrete subclasses.
        Circle myCircle = new Circle("Blue", 5.0);
        Rectangle myRectangle = new Rectangle("Green", 4.0, 6.0);

        // Call inherited concrete methods
        myCircle.display();
        myRectangle.display();

        // Call implemented abstract methods
        System.out.println("Color of Circle: " + myCircle.getColor());
        System.out.println("Area of Circle: " + myCircle.calculateArea());

        System.out.println("\nColor of Rectangle: " + myRectangle.getColor());
        System.out.println("Area of Rectangle: " + myRectangle.calculateArea());
    }
}
```

In this example:

- `Shape` is **abstract**. It defines the concept that a shape must have a color and a way to calculate its area, but it doesn't know *how* to calculate the area for a generic shape.
- `calculateArea()` is an **abstract method**. It forces subclasses like `Circle` and `Rectangle` to provide their own specific formulas.
- `Circle` and `Rectangle` are **concrete** because they provide implementations for all abstract methods they inherit.

This perfectly demonstrates abstraction by hiding the specific area calculation formulas inside the respective shape classes and providing a common, simple method name (`calculateArea`) to get the result.
