ch_01_Classes_and_Objects

April 21, 2024

1 OOPs

Object-Oriented Programming (OOP) is a programming paradigm that uses objects, which are instances of classes, to design and organize code. The fundamental idea behind OOP is to model real-world entities and their interactions in a way that is both intuitive and efficient. Here are some key principles and reasons why OOP is widely used:

1. **Encapsulation:**
   - *Principle:* Encapsulation refers to the bundling of data (attributes) and methods that operate on the data within a single unit (class).
   - *Why:* It helps in hiding the internal details of how an object works and exposes only what is necessary. This makes the code more modular and reduces the likelihood of unintended interference.
2. **Abstraction:**
   - *Principle:* Abstraction involves simplifying complex systems by modeling classes based on the essential properties and behaviors they share.
   - *Why:* Abstraction allows developers to focus on the essential features of an object while ignoring the non-essential details. It makes the code more understandable and scalable.
3. **Inheritance:**
   - *Principle:* Inheritance allows a class (subclass/child) to inherit properties and behaviors from another class (superclass/parent).
   - *Why:* It promotes code reusability by allowing common functionalities to be defined in a superclass and shared by multiple subclasses. This reduces code duplication and makes maintenance easier.
4. **Polymorphism:**
   - *Principle:* Polymorphism allows objects of different classes to be treated as objects of a common base class. It comes in two forms: compile-time (method overloading) and runtime (method overriding).
   - *Why:* Polymorphism enables flexibility and adaptability in code. It allows you to write code that can work with objects of various types without knowing their specific class, making the code more extensible.
5. **Modularity:**
   - *Principle:* OOP promotes the organization of code into modular units (classes), each responsible for a specific functionality.
   - *Why:* Modularity enhances code readability, maintainability, and reusability. Changes to one part of the code are less likely to affect other parts if the code is well-modularized.
6. **Code Reusability:**
   - *Principle:* OOP encourages the reuse of existing code through inheritance and compo-

sition.
    - *Why:* Code reusability reduces development time and effort. Instead of writing new code from scratch, developers can build upon existing, tested components.
7. **Ease of Maintenance:**
    - *Why:* OOP makes it easier to maintain and update code. Changes can be localized to specific classes, and the overall structure of the program remains more stable.
8. **Real-world Modeling:**
    - *Why:* OOP allows developers to model their software based on real-world entities, making the design more intuitive and reflective of the problem domain.

In summary, OOP provides a set of principles and practices that lead to more organized, modular, and maintainable code. It aligns well with how we naturally think about and model the real world, making it a powerful paradigm for designing software systems.

2 Table Of Contents:

To become proficient in object-oriented programming (OOP) in Java, it's important to delve into various topics. Here's a list of key topics you should learn in detail:

1. **Classes and Objects:**
    - Understand the concept of classes and objects.
    - How to define classes and create objects.
    - Constructors and their role in object initialization.
    - Static members and methods.
2. **Encapsulation:**
    - Access modifiers (public, private, protected, default).
    - Getter and setter methods.
    - Understanding and implementing encapsulation to hide internal details.
3. **Inheritance:**
    - The concept of inheritance and its advantages.
    - Syntax for extending a class.
    - Overriding methods and using the `super` keyword.
    - The difference between classes and interfaces.
4. **Polymorphism:**
    - Understanding polymorphism in terms of method overloading and method overriding.
    - Compile-time and runtime polymorphism.
    - The `instanceof` operator.
5. **Abstraction:**
    - Abstract classes and methods.
    - Interfaces and their implementation.
    - Abstract classes vs. interfaces.
    - How abstraction contributes to code design.
6. **Interfaces:**
    - Defining and implementing interfaces.
    - Default and static methods in interfaces.
    - Multiple inheritance through interfaces.
    - Functional interfaces and lambda expressions.
7. **Packages:**

- Creating and organizing code using packages.
- Accessing classes from different packages.
- The `import` statement.

8. **Composition:**
   - Using composition to create complex objects.
   - Relationship between classes through composition.
   - Advantages of composition over inheritance.
9. **Enum Types:**
   - Defining and using enumerated types.
   - Enum methods and attributes.
   - Use cases for enums.
10. **Exception Handling:**
    - The `try`, `catch`, `finally` blocks.
    - Creating custom exceptions.
    - Exception hierarchy and common exception classes.
11. **Lambda Expressions:**
    - Introduction to functional programming in Java.
    - Syntax and usage of lambda expressions.
    - Functional interfaces and the `@FunctionalInterface` annotation.
12. **Stream API:**
    - Working with streams to process collections.
    - Intermediate and terminal operations.
    - Collectors and reduction.
13. **Design Patterns:**
    - Understanding common design patterns (e.g., Singleton, Factory, Observer).
    - Implementing and recognizing design patterns in Java.
14. **Garbage Collection:**
    - How Java manages memory through garbage collection.
    - The `finalize` method and its limitations.
    - Best practices for memory management.
15. **Reflection:**
    - Using reflection to inspect and manipulate classes.
    - `Class` class and its methods.
    - Limitations and considerations.
16. **Concurrency:**
    - Introduction to multi-threading.
    - Synchronization and locks.
    - `volatile` keyword and atomic operations.
17. **Annotations:**
    - Understanding and creating annotations.
    - Built-in annotations in Java.
    - Annotation processors and their use.
18. **Java 8 Features:**
    - New features introduced in Java 8 (e.g., Streams, Lambda Expressions, Optional).
    - Understanding and using these features in practical scenarios.

By thoroughly covering these topics, you'll have a solid foundation in object-oriented programming with Java, enabling you to build robust and maintainable software applications.

3 Class and Object

In Java, a class is a blueprint or a template for creating objects. An object, on the other hand, is an instance of a class. Let's break down these concepts:

1. **Class:**
   - **Definition:** A class is a blueprint or a prototype that defines the attributes and behaviors common to all objects of that type.
   - **Purpose:** It encapsulates the data (attributes) and methods (functions) that operate on that data into a single unit.
   - **Example:**
   ```
   public class Car {
       // Attributes
       String brand;
       int year;

       // Methods
       void start() {
           System.out.println("The car is starting.");
       }
   }
   ```
   - In this example, `Car` is a class that defines a blueprint for cars. It has attributes like `brand` and `year` and a method `start` that describes how a car starts.
2. **Object:**
   - **Definition:** An object is an instance of a class. It is a tangible and concrete entity created from the blueprint provided by the class.
   - **Purpose:** Objects represent real-world entities and allow us to work with specific instances of a class.
   - **Example:**
   ```
   public class Main {
       public static void main(String[] args) {
           // Creating an object of the Car class
           Car myCar = new Car();

           // Setting values for the attributes
           myCar.brand = "Toyota";
           myCar.year = 2022;

           // Calling a method on the object
           myCar.start();
       }
   }
   ```
   - In this example, `myCar` is an object of the `Car` class. It has attributes like `brand` and `year`, and you can perform actions on it, such as calling the `start` method.

In summary, a class is a blueprint that defines the structure and behavior of objects, and an object is an instance of a class, representing a specific occurrence of the entity described by the class. Classes provide a way to structure code, and objects allow you to work with and manipulate data based on that structure.

[ ]:
```java
public class Car {
    // Attributes
    String brand;
    int year;

    // Methods
    void start() {
        System.out.println("The car is starting.");
    }
}

Car myCar = new Car();

myCar.start()
```

```
The car is starting.
```

[ ]:
```java
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Car class
        Car myCar = new Car();

        // Setting values for the attributes
        myCar.brand = "Toyota";
        myCar.year = 2022;

        // Calling a method on the object
        myCar.start();
    }
}

Main m= new Main();
m.main(null);
```

```
The car is starting.
```

4 Instance and Reference

When you create an object in Java, both a reference and an instance are created.

1. **Instance:**
   - The "instance" refers to the actual object that is created in memory. It occupies a specific location in the computer's memory and holds the data defined by the class (attributes/fields).
   - In the context of your `Car` class, when you write `Car myCar = new Car();`, the `new Car()` part creates an instance of the `Car` class, representing an actual car in memory.
2. **Reference:**

- The "reference" is a variable that points to the memory location of the instance. It's a way for you to access and work with the object. In the same `Car` example, `myCar` is a reference to the newly created `Car` instance.
- The reference allows you to interact with the instance by calling its methods or accessing its attributes.

Let's break down the process of object creation:

```
Car myCar = new Car();
```

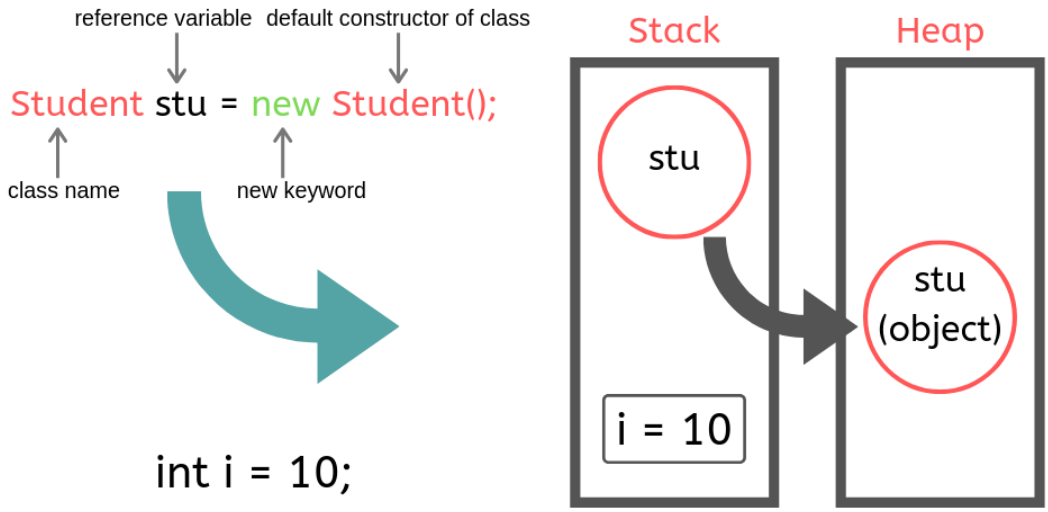
1. `new Car()`: This part of the statement creates a new instance of the `Car` class. It allocates memory for the attributes defined in the class and initializes them with their default values (e.g., `null` for `String`, `0` for `int`).
2. `Car myCar`: This part declares a reference variable named `myCar` of type `Car`. It is assigned the memory address of the newly created `Car` instance.

So, when you create an object, you are essentially creating an instance of a class in memory, and you use a reference to interact with and manipulate that instance.

Regarding the "region behind this creation," it's important to note that Java uses a combination of the heap and the stack for memory management:

- **Heap:** The actual objects (instances) are created in the heap memory, which is a region used for dynamic memory allocation. Objects in the heap persist as long as they are reachable (referenced) from a live thread.
- **Stack:** The references to objects are stored in the stack memory. When you declare a reference variable (like `Car myCar`), the reference is stored in the stack, pointing to the actual object in the heap.

Understanding this memory allocation is crucial for managing memory effectively and avoiding memory leaks. The Java Virtual Machine (JVM) takes care of garbage collection, which automatically reclaims memory occupied by objects that are no longer reachable, ensuring efficient memory usage.



5 Let's Understand Of Concept of Stack and Heap...

In object-oriented programming (OOP), the concepts of heap and stack are crucial for understanding how memory is managed during the creation and execution of objects and methods. Let's break down the concepts of heap and stack in the context of OOP, specifically in Java:

5.0.1 Heap:

1. **Object Creation:**
   - When you create an object using the `new` keyword, memory is allocated on the heap to store the object's data (attributes).
   - Example: `Car myCar = new Car();` creates a `Car` object on the heap.
2. **Lifetime:**
   - Objects in the heap have a dynamic lifetime. They persist as long as they are reachable, meaning there is at least one reference pointing to them.
3. **Data Storage:**
   - The actual data (attributes and instance variables) of an object are stored in the heap. This is where the object's state is maintained.
4. **Garbage Collection:**
   - The Java Virtual Machine (JVM) has a garbage collector that automatically identifies and collects objects that are no longer reachable, freeing up memory for future use.

5.0.2 Stack:

1. **Reference Variables:**
   - When you declare a reference variable (e.g., `Car myCar;`), the reference itself is stored on the stack.
   - Example: `Car myCar = new Car();` creates a reference variable `myCar` on the stack.
2. **Method Execution:**
   - When a method is called, a new frame is created on the stack to store local variables, method parameters, and control flow information.
   - The method's local variables (including references to objects) are stored in this stack frame.
3. **Method Invocation:**
   - When a method is invoked, a new stack frame is pushed onto the stack to handle the method's execution.
   - Local variables and method parameters are pushed onto this frame.
4. **LIFO (Last In, First Out):**
   - The stack follows the Last In, First Out (LIFO) principle. The last item pushed onto the stack is the first one to be popped off.

5.0.3 Step-by-Step Process:

1. **Object Creation:**
   - `Car myCar = new Car();` is executed.
   - The `new Car()` part allocates memory on the heap for a new `Car` object.
2. **Reference Variable:**
   - `Car myCar` is a reference variable declared on the stack.
   - The memory address of the newly created `Car` object on the heap is assigned to `myCar`.

3. **Method Invocation:**
    - If the `Car` class has methods, when a method is called (`myCar.start()`), a new stack frame is created for that method's execution.
    - Local variables (including references) for the method are stored in this stack frame.
4. **Lifetime Management:**
    - The lifetime of the reference variable (`myCar`) on the stack is determined by its scope (e.g., method scope, class scope).
    - The lifetime of the object on the heap is determined by its references. When there are no more references to the object, it becomes eligible for garbage collection.

Understanding the interplay between the heap and the stack is essential for effective memory management in Java and other OOP languages. The stack handles method calls and local variables, while the heap stores the actual object data and dynamically allocated memory.

6 Why Heap and Stack?

The use of heap and stack in programming languages, including object-oriented ones like Java, is based on the need for efficient memory management and the characteristics of each memory region. Each has its own purpose and benefits, contributing to the overall performance and reliability of a program.

6.0.1 Stack:

1. **Fast Access:**
    - The stack is a region of memory that provides fast access to variables.
    - It uses a Last In, First Out (LIFO) structure, making it efficient for method calls and local variable storage.
2. **Automatic Memory Management:**
    - Memory allocated on the stack is automatically reclaimed when a method completes its execution. This is due to the LIFO structure, where the most recently allocated memory is the first to be deallocated.
3. **Static Memory Allocation:**
    - Memory allocation on the stack is static and deterministic. The size and lifetime of variables are known at compile time.
4. **Limited Size:**
    - The stack has a limited size, typically smaller than the heap. This limitation makes it suitable for managing local variables and controlling the depth of method calls.

6.0.2 Heap:

1. **Dynamic Memory Allocation:**
    - The heap is a region of memory that allows for dynamic memory allocation.
    - Memory can be allocated and deallocated at runtime, providing flexibility for managing objects with varying lifetimes.
2. **Object Storage:**
    - Complex data structures like objects and arrays are stored in the heap. These structures may have unknown sizes or lifetimes.
3. **Variable Size:**

- The heap allows for the allocation of memory with variable sizes. This is useful when dealing with data structures that can grow or shrink during program execution.

4. **Longer Lifetime:**
   - Objects in the heap can have a longer lifetime than those on the stack. They persist as long as there are references to them, making the heap suitable for managing objects with extended lifetimes.

6.0.3 Why Both?

1. **Optimized Memory Usage:**
   - Combining the stack and heap allows for optimized memory usage. The stack efficiently manages local variables and method calls, while the heap handles dynamic memory allocation for more complex data structures.
2. **Automatic and Explicit Memory Management:**
   - The stack offers automatic memory management, while the heap provides explicit control over memory allocation and deallocation. This combination strikes a balance between simplicity and flexibility.
3. **Performance:**
   - The use of both heap and stack contributes to overall program performance. Fast access to local variables and efficient memory allocation for dynamic data structures enhance the program's responsiveness.

In summary, the use of heap and stack in programming languages is a design choice that balances factors such as speed, memory efficiency, and flexibility. Each memory region serves a specific purpose, and their combined usage allows for effective and optimized memory management in a wide range of scenarios.

7 Static Member and Method

7.1 Data Member

In object-oriented programming (OOP), a data member is a variable that is associated with an object or a class. Data members represent the state or attributes of objects and are used to store information that describes the characteristics of the object. These variables define the object's properties and contribute to its behavior.

There are two main types of data members in OOP: instance variables (also known as non-static variables) and class variables (also known as static variables).

7.1.1 1. Instance Variables:

- **Definition:** Instance variables are associated with instances of a class. Each object (instance) of the class has its own copy of these variables, and they represent the unique state of each object.
- **Declaration:** Declared within the class but outside any method, using the syntax: `accessModifier dataType variableName;`
- **Example:** `java public class Car { // Instance variables String brand; int year; }`
- In this example, `brand` and `year` are instance variables of the `Car` class.

7.1.2 2. Class Variables (Static Variables):

- **Definition:** Class variables are associated with the class itself rather than with instances. There is only one copy of a class variable shared among all instances of the class.
- **Declaration:** Declared with the `static` keyword, using the syntax: `accessModifier static dataType variableName;`
- **Example:** `java public class MathOperations { // Class variable static final double PI = 3.14; }`
- In this example, `PI` is a class variable of the `MathOperations` class.

7.1.3 Use Cases for Data Members:

1. **Object State:**
   - Instance variables are used to represent the state or attributes of objects.
2. **Object Initialization:**
   - Instance variables are often initialized in the constructor when objects are created.
3. **Shared Information:**
   - Class variables are used to represent shared information that applies to all instances of a class.
4. **Constants:**
   - Class variables can be used to define constants that are shared among all instances.
5. **Singleton Pattern:**
   - Class variables are used in the implementation of the Singleton design pattern to ensure a single instance of a class.

7.1.4 Important Considerations:

- Instance variables contribute to the unique state of each object, while class variables are shared among all instances.
- The access modifiers (`public`, `private`, `protected`, or default) determine the visibility and accessibility of data members.
- It is a good practice to encapsulate data members by providing getter and setter methods to control access and modification.

In summary, data members in OOP are variables that define the properties or state of objects. They play a crucial role in representing the characteristics of objects and are essential for modeling real-world entities in an object-oriented system.

8 Static V/s Non-Static

let's differentiate between static and non-static variables using a table format with rows and columns:

Property	Static Variable	Non-Static Variable (Instance Variable)
Storage Location	Stored in the class area of the memory (one copy)	Each instance of the class has its own copy
Access	Accessed using the class name (e.g., `ClassName.variable`)	Accessed using an instance of the class (e.g., `object.variable`)

Property	Static Variable	Non-Static Variable (Instance Variable)
Memory Allocation	Allocated once when the class is loaded into memory	Allocated each time an object is created
Initialization Timing	Initialized when the class is loaded	Initialized when an object is created
Usage Examples	Constants, shared information, counters, etc.	Object properties, state, unique information, etc.
Keyword	Declared with the `static` keyword	Declared without the `static` keyword
Visibility	Shared among all instances and accessible globally	Limited to the specific instance and its scope

Example:

```
public class MyClass {
    // Static variable
    static int staticVariable;

    // Non-static (instance) variable
    int instanceVariable;

    // Static method accessing static variable
    static void staticMethod() {
        staticVariable = 10;  // Valid
        // instanceVariable = 5;  // Invalid - Non-static variable cannot be accessed in a sta
    }

    // Non-static method accessing both static and non-static variables
    void instanceMethod() {
        staticVariable = 20;  // Valid
        instanceVariable = 15;  // Valid
    }
}
```

In the example above, `staticVariable` is a static variable, while `instanceVariable` is a non-static (instance) variable. The `staticMethod` is a static method that can access only static variables directly. The `instanceMethod` is a non-static method that can access both static and non-static variables directly.

9 Thank You!