Access Modifiers

In Java, access modifiers define the scope or visibility of a class, method, variable, or constructor. Here are the four main types:

**Public (public)**:

* Accessible from any other class.
* Used when you want your class, method, or variable to be available universally across all classes and packages.

**Protected (protected)**:

* Accessible within the same package and by subclasses (even if they are in different packages).
* Typically used when designing a base class for inheritance.

**Default (Package-Private)** (no keyword):

* If no modifier is specified, the default access level is applied, making it accessible only within its own package.
* Useful when you want a class or member to be package-scoped but hidden from classes outside the package.

**Private (private)**:

* Accessible only within the class where it is declared.
* This is the most restrictive level and is often used to encapsulate data or methods within a class.

A box with check marks and red x

Description automatically generated

A black box with white text

Description automatically generated

A white background with black text

Description automatically generated

Non-Access Modifiers

A white and black text on a white background

Description automatically generated

Static Keyword:

The static keyword in Java is primarily used to create members (variables and methods) that belong to the class itself, rather than to specific instances of the class. This means that these static members can be accessed directly using the class name, without creating an object of the class.

**Static variable:**A variable marked as static belongs to the **class** rather than an instance of the class. It is shared among all instances of the class.

**Static Method:**

A static method belongs to the **class** and can be called without creating an instance of the class. Static methods can only access static data (variables) and call other static methods.

**Static Blocks:**  
A static block is used to initialize static variables when the class is first loaded into memory. This block runs **once** when the class is loaded, before any object is created or any static method is called.

A screenshot of a computer code

Description automatically generated

Final Keyword:

The final keyword in Java is used to declare entities that cannot be modified after they are initialized. This means that their values or behaviors remain fixed throughout the program's execution.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Abstract Keyword:

In Java, the abstract keyword is used to declare **abstract classes** and **abstract methods**, and it plays a crucial role in supporting **abstraction** in object-oriented programming. Abstraction allows you to define the structure of a class or method without providing the complete implementation.

=====================================================================

POLYMORPHISM

is the ability of a method to take on different behaviors depending on the object that invokes it. This allows a single interface to represent a broader range of actions or operations, where the specific behavior is determined by the type of the object at runtime. Polymorphism enables a unified interface for a general set of actions, allowing methods to be defined in a superclass and overridden by subclasses to provide specialized behavior.

A screen shot of a computer

Description automatically generated

**Summary of the Workflow**

* **Polymorphic Behavior**: The program uses polymorphism to call the appropriate processPayment() method depending on the actual type of each Payment object. Each subclass (CashPayment, CardPayment, MobileWalletPayment) provides its own specific implementation of processPayment().
* **Runtime Decision**: The correct processPayment() method is chosen at runtime based on the object’s actual type, even though all objects are referenced as type Payment.
* **Extensibility**: This design allows new payment types to be added (e.g., CryptoPayment) with minimal changes to existing code, as each new class would implement its own processPayment() method.

**Benefits**

* **Polymorphism** makes the code flexible, allowing Payment objects to handle various payment methods without needing to alter the main application logic.

Method Overriding/ Method Overloading

Refer notes.

Upcasting/Downcasting

Refer notes.

==============================================

ENCAPSULATION

Encapsulation is a core principle in which data and associated methods are bundled within a class, exposing only select access through controlled getter and setter methods. This approach ensures data integrity, security, and validation, serving as a protective layer that shields internal data from direct external manipulation.

A screen shot of a computer

Description automatically generated

A screen shot of a computer screen

Description automatically generated



A screenshot of a computer program

Description automatically generated

**Workflow:**

1. **Customer Creation**: A Customer object is created with name and phoneNumber, both encapsulated.
2. **Order Creation**: An Order is created with a list of items (encapsulated).
3. **Payment Processing**: The payment details are processed and encapsulated in the Payment class. Payment cannot be changed once it is set, protecting the integrity of the transaction.
4. **Secure Access**: Only necessary data is exposed through getter methods. Sensitive details like payment.amount are not mutable directly and are only processed securely within the class.

**Benefits of This Approach:**

* **Data Protection**: Sensitive data like payment details are encapsulated in the Payment class and cannot be changed directly from outside the class, reducing the risk of errors or unauthorized changes.
* **Controlled Modification**: Setters are used where modification is allowed (e.g., name, phoneNumber), but immutable fields (e.g., amount in Payment) are protected.
* **Flexibility**: The system allows for payment methods to be processed securely and ensures that only the right methods are exposed to the user or external systems.

==============================================

ABSTRACTION

Abstraction is a key programming concept that focuses on presenting only essential information while hiding unnecessary details. Its goal is to simplify complex systems by exposing only the relevant aspects, thus enhancing clarity and reducing complexity. Abstraction helps developers manage and work with complex data more efficiently by revealing only what’s necessary.

In Java, abstraction can be achieved using two mechanisms:

**Abstract classes:** These serve as base classes that cannot be instantiated. They can include abstract methods (declared without implementation), regular methods, constructors, constants, instance variables and so on. Subclasses inherit from abstract classes and provide implementations for any abstract methods.

In Java, abstract classes can include the following elements:

* **Constructors**: Abstract classes can have constructors, but they cannot be used to instantiate the abstract class itself. Instead, these constructors are called when a concrete subclass is created, allowing initialization of common fields or performing setup logic.
* **Concrete Methods**: These are regular methods with complete implementations. Concrete methods in abstract classes allow shared functionality to be provided to all subclasses without requiring them to implement the method.
* **Abstract Methods**: These methods are declared without an implementation (i.e., without a method body) and must be implemented by any non-abstract subclass. Abstract methods define behaviors that subclasses are expected to provide specific implementations for.
* **Instance Variables (Fields)**: Abstract classes can have instance variables, which are typically used to store the state or common data for subclasses. These can be private, protected, or public, depending on the access requirements.
* **Static Variables (Static Fields)**: Abstract classes can have static variables, which are shared among all instances of the class, regardless of the subclass.
* **Static Methods**: Abstract classes can also contain static methods. These methods belong to the class itself rather than any instance and can be accessed directly using the abstract class name.
* **Final Methods**: Although rare, abstract classes can have final methods, which cannot be overridden by subclasses. This can be useful when a specific behavior in the abstract class should not be modified.
* **Nested Classes or Interfaces**: Abstract classes can contain nested (inner) classes or interfaces. This allows structuring related functionality within the abstract class without making it publically accessible outside.
* **Constants**: Abstract classes can define constants (typically public static final variables) that can be accessed by subclasses directly.

**Interfaces:** An interface is a contract that specifies a set of methods that any implementing class must provide.

* **Abstract Methods:** By default, all methods in an interface are abstract and must be implemented by any class that implements the interface.
* **Default Methods:** Java 8 introduced default methods, which are methods with an implementation in an interface. These allow the interface to provide a default behavior that implementing classes can override if needed.
* **Static Methods:** Interfaces can also contain static methods, which belong to the interface itself and can be called without an instance of the interface. These methods cannot be overridden by implementing classes.
* **Constants (Static Final Fields):** Fields in an interface are implicitly public, static, and final, meaning they are constants. Interfaces cannot contain instance variables.
* **Multiple Inheritance Support**: Unlike classes, a Java class can implement multiple interfaces, allowing it to inherit behaviors from multiple sources.
* **Public Access by Default**: All methods in an interface are implicitly public, and access modifiers are not required. Methods in interfaces cannot be private or protected.
* **No Constructors**: Interfaces cannot have constructors, as they cannot be instantiated directly. Classes implementing the interface provide the actual instantiation.
* **Functional Interfaces:** Java 8 introduced functional interfaces—interfaces with exactly one abstract method. These are used for lambda expressions and functional programming styles.
* **Inheritance Among Interfaces**: An interface can extend one or more other interfaces, allowing a new interface to inherit methods from other interfaces, even though it doesn’t support inheritance from classes.

A screenshot of a computer screen

Description automatically generated

**================================================================**

**Super Keyword:** The super keyword in Java allows access to members (methods and variables) of the immediate parent class, providing these key functions:

* **Access Parent Class Members**: super.fieldName or super.methodName() can be used to access fields and methods from the parent class, useful for overridden members.
* **Invoke Parent Constructor**: super() calls the parent’s constructor and must be the first line in the subclass’s constructor, ensuring the parent class initializes before the child.
* **Use in Overridden Methods**: In an overridden method, super.methodName() can call the parent’s version of the method.
* **Not Usable in Static Contexts**: super applies only to instances and can’t be used in static methods or blocks.

==============================================

This Keyword

The **this** keyword in Java refers to the current instance of a class and has several key uses:

1. **Differentiates Variables**: Used to distinguish instance variables from parameters with the same name.
2. **Constructor Chaining**: this() calls another constructor in the same class to avoid redundancy.
3. **Passing Current Instance**: Allows the current object to be passed to methods or constructors.
4. **Method Chaining**: Enables methods to return the current instance for chaining.
5. **Accessing Outer Class in Inner Classes**: Refers to the instance of the outer class using OuterClass.this.

==============================================

INHERITANCE

Inheritance allows a subclass to acquire the properties and behaviors of a superclass, enabling code reuse and establishing a clear class hierarchy.

A screenshot of a computer screen

Description automatically generated