**The Core Concepts of OOP 📚**

OOP is a programming paradigm that structures software around **objects**, which are real-world entities that have **state** (attributes) and **behavior** (methods). This approach simplifies development and maintenance by organizing code into reusable, modular components. The main concepts of OOP are Class, Object, Inheritance, Polymorphism, Abstraction, and Encapsulation.

**1. Class and Object 👥**

* **Class**: A **class** is a blueprint, or a template, for creating objects. Think of it as a factory for objects. It defines the attributes and methods that every object of that class will possess. For example, a Student class defines what it means to be a student, including properties like name, rollNo, and section.
* **Object**: An **object** is an instance of a class. It's a concrete entity created from the class blueprint. Each object has its own unique set of attribute values. student1, student2, and student3 are all objects of the Student class.

**2. The Four Pillars of OOP 🏛️**

These are the fundamental principles that make OOP so powerful.

* **Inheritance** 🧬: Inheritance allows a new class (the **child** or **derived** class) to reuse the attributes and methods of an existing class (the **parent** or **base** class). This promotes code reuse and establishes a hierarchical "is-a" relationship (e.g., a Student **is a** Person).
* **Polymorphism** 🎭: The term means "many forms." Polymorphism enables a single entity (like a method) to have multiple forms. It is implemented in two ways:
  + **Method Overloading**: Defining multiple methods with the same name but different parameters within a single class.
  + **Method Overriding**: A child class providing a specific implementation for a method that is already defined in its parent class.
* **Abstraction** 🧠: Abstraction is the principle of hiding complex internal details and showing only the necessary functionalities. It focuses on what an object does rather than how it does it. A car's engine is hidden from the user, but the user can interact with the startEngine() method. Abstraction is achieved through **abstract classes** and **interfaces**.
* **Encapsulation** 📦: Encapsulation is the practice of bundling data (attributes) and the methods that operate on that data into a single unit (the class). It's a way of protecting data by restricting direct access. Attributes are often made private and accessed only through public methods (getters and setters), ensuring data integrity.

**Advantages of OOP in Java ✅**

* **Simpler Implementation**: OOP concepts are straightforward to apply in Java.
* **Code Reusability**: Inheritance and other concepts allow for reusing existing code, reducing development time.
* **Faster Execution**: Programs based on the OOP paradigm often execute more quickly than procedural programs.
* **Data Hiding**: Encapsulation helps to keep sensitive data private and secure from unauthorized access.

**📝 CLASSES & OBJECTS :**

In Java, **classes** and **objects** are foundational to the Object-Oriented Programming (OOP) paradigm. This approach models software around real-world entities, or objects, which have distinct **states** (data) and **behaviors** (methods).

**1. Classes: The Blueprint 🏛️**

A **class** is a blueprint or template for creating objects. It defines the structure and functionality that all objects of that class will share. A class itself doesn't take up any memory; it's a logical construct that outlines the data members and methods.

**Class Components 🧱**

* **Data Members (Variables)**: These store an object's state. There are three types:
  + **Instance variables:** Unique to each object and declared within a class but outside any method.
  + **Class variables:** Shared by all objects of a class and declared with the static keyword.
  + **Local variables:** Temporary variables defined within a method, constructor, or block.
* **Methods**: These define an object's behavior. They are used to manipulate the data members.
* **Constructors**: Special methods that initialize a new object when it's created.

**Creating a Class ✍️**

A class is declared using the class keyword followed by its name. The syntax is: access\_modifier class ClassName { ... }.

**2. Objects: The Instance 🧍**

An **object** is a concrete instance of a class. It is a variable of a class type that exists in memory. An object's state is its unique set of values for the instance variables defined in its class.

**Creating an Object 🆕**

Objects are created from a class using the **new** keyword. The process involves three steps:

1. **Declaration**: A variable is declared with the object type (e.g., Dog obj;).
2. **Instantiation**: Memory is allocated for the object using the new keyword.
3. **Initialization**: A constructor is called to set the initial values of the object's data members.

**Accessing Members ➡️**

Once an object is created, you can access its instance variables and methods using the dot (.) operator: objectName.variable or objectName.method().

**3. Java Source File Rules 📜**

These rules ensure proper compilation and execution of Java code.

* A Java source file can have **only one public class**.
* The source file name must match the public class name (e.g., EmployeeTest.java for public class EmployeeTest).
* The package statement must be the first line of the file.
* import statements must be placed after the package statement.
* These rules apply to all classes within a single source file.

For example, Employee.java and EmployeeTest.java demonstrate how a main method in a separate test class creates and uses objects of another class.

**Hands-on Practice: Class and Object**

**Objective:** Create a Book class and then create objects to represent different books.

1. **Create a Book Class:**
   * Create a file named Book.java.
   * Inside the file, define a public class called Book.
   * Give the class the following **instance variables**:
     + String title;
     + String author;
     + int pages;
   * Add a **constructor** that takes title, author, and pages as parameters and initializes the instance variables using the this keyword.
   * Add a **method** called printBookDetails() that prints the book's title, author, and number of pages to the console.
2. **Create a Library Class to Test It:**
   * Create a separate file named Library.java.
   * Inside this file, create a public class called Library with a main method.
   * In the main method, create **two new Book objects** using the new keyword and the constructor you defined. For example, Book book1 = new Book("The Hobbit", "J.R.R. Tolkien", 310);.
   * Call the printBookDetails() method for each of the two book objects you created.
   * Compile and run the Library.java file to see the output.
3. class Book{
4. String title;
5. String author;
6. int pages;
7. Book(){
8. }
9. Book(String title, String author, int pages){
10. this.title = title;
11. this.author = author;
12. this.pages = pages;
13. }
14. public void printBookDetails(){
15. System.out.println(title+" "+author+" "+pages);
16. }
17. }
18. class Library{
19. public static void main (String  [] args){
20. Book obj =new Book();
21. Book obj1 = new Book("Python","Mohith",1000);
22. Book obj2 = new Book("Java","Arnav",200);
23. obj1.printBookDetails();
24. obj2.printBookDetails();
25. obj.printBookDetails();
26. }
27. }

**Java Class Attributes :**

**Java Class Attributes 📝**

In Java, **class attributes** (also known as fields or variables) are the data that define a class's state. They are declared within the class scope and hold values that describe the objects created from that class.

**Declaring and Accessing Attributes**

To declare a class attribute, you specify an **access modifier** (e.g., public, private), a **data type**, and the attribute's name.

**Example:**

Java

public class Dog {

String breed; // An instance variable or attribute

int age; // Another instance variable

}

To access an attribute, you must first create an object of the class using the new keyword. Then, you use the **dot (.) operator** followed by the attribute's name.

**Syntax:** object\_name.attribute\_name

Java

Dog myDog = new Dog();

System.out.println(myDog.breed);

**Modifying and Restricting Attributes**

You can change an attribute's value by using the dot operator with the **assignment operator (=)**.

**Syntax:** object\_name.attribute\_name = new\_value;

Java

Dog myDog = new Dog();

myDog.age = 3; // Modifying the 'age' attribute

To prevent an attribute's value from being changed after it's been initialized, you can make it **read-only** by adding the **final keyword** during its declaration. If you try to modify a final attribute, the compiler will produce an error.

**Syntax:** access\_modifier final data\_type attribute\_name;

Java

class Dog {

final String name = "Tommy";

}

// In another class:

Dog myDog = new Dog();

// myDog.name = "Spike"; // This would cause a compilation error

**Objective:** Define a class with attributes and access their values.

1. **Create a Car Class:**
   * Create a file named Car.java.
   * Inside the file, define a class called Car with the following **instance variables** (attributes):
     + String brand;
     + String model;
     + int year;
   * Add a **constructor** that takes brand, model, and year as parameters to initialize these attributes.
2. **Create a Garage Class to Test It:**
   * Create a separate file named Garage.java.
   * Inside this file, create a public class Garage with a main method.
   * In the main method, create a new Car object using the constructor. For example: Car myCar = new Car("Toyota", "Camry", 2022);.
   * Use System.out.println() to print the values of the brand, model, and year attributes by accessing them with the dot operator, like myCar.brand.
   * Compile and run Garage.java to see the output.

**2. Modifying Attributes 🔄**

**Objective:** Change the values of an object's attributes after it has been created.

1. **Modify the Car Class:**
   * Use the same Car.java and Garage.java files from the previous exercise.
   * In the main method of the Garage class, after creating your myCar object, add new lines of code.
   * **Modify** one or more attributes. For example: myCar.year = 2023;.
   * Print the values again to see the changes. Your output should show the new year.
2. class Car{
3. String brand;
4. String model;
5. int year;
6. Car(){
7. }
8. Car(String brand, String model, int year){
9. this.brand = brand;
10. this.model = model;
11. this.year = year;
12. }
13. }
14. public class Garage {
15. public static void main(String[] args) {
16. Car myCar = new Car("Tata","nano",2020);
17. myCar.year = 2022;
18. System.out.println(myCar.brand);
19. System.out.println(myCar.year);
20. }
21. }

**Java Class Methods :**

Class methods are functions defined within a class that perform operations and can access or modify the class's attributes. They are a fundamental part of an object's behavior.

**Creating and Accessing Methods**

To **declare** a method, you specify its modifier, returnType, nameOfMethod, and Parameter List within a class. For example, a method to find the minimum of two numbers would be declared as public int minimum(int n1, int n2).

To **access** a public class method, you must first create an **object** of the class. You then use the **dot (.) operator** followed by the method name and any required parameters.

**Syntax:** object\_name.method\_name([parameters]);

Java

// Example of accessing a method

Util util = new Util();

int c = util.minimum(a, b); // Accessing the minimum() method

**The this Keyword**

The **this** keyword is a reference to the **current object** of a class. It can be used within an instance method or a constructor to refer to the class's members.

**this is used to:**

* **Differentiate between instance variables and local variables** that have the same name. By using this.variableName, you explicitly refer to the instance variable.
* **Invoke one constructor from another** within the same class, a practice known as **explicit constructor invocation**.

**Public vs. Static Methods**

Class methods can be either **public** or **static**. The key difference lies in how you access them:

* **Public Methods** (public void fun2()): These methods **require an object** of the class to be accessed. You must create an instance of the class and then use the object's reference to call the method.
* **Static Methods** (static void fun1()): These methods **belong to the class itself**, not to any specific object. You can call a static method directly using the class name, without needing to create an object. This is useful for utility functions that don't depend on an object's state.

**The finalize() Method**

The finalize() method is a special method that can be defined in a class. The Java runtime's **garbage collector** calls it just before an object is permanently destroyed. It provides an opportunity to perform clean-up operations, such as closing open files, before the object is removed from memory. The method has the signature protected void finalize(). However, its execution is not guaranteed, and you should not rely on it for critical resource management.

**Java Methods :**

Here is an enhanced and efficient summary of Java methods for your notes.

**Java Methods: An Overview 🛠️**

A **method** is a block of code that performs a specific operation. It's a fundamental part of Java programming, allowing you to organize code into reusable and manageable units. Methods are defined within a class.

**Method Signature & Structure**

A method's **signature** consists of its name and parameter list. The full declaration includes:

* **modifier**: Specifies the access type (e.g., public, private).
* **returnType**: The data type of the value the method returns. Use void if the method doesn't return anything.
* **nameOfMethod**: The name of the method.
* **Parameter List**: A comma-separated list of variables passed into the method. This list can be empty.
* **method body**: The code block containing the statements the method executes.

Java

public static int minFunction(int n1, int n2) {

// method body

return result;

}

**Key Concepts 🔑**

**1. Calling a Method**

To use a method, you **invoke** or **call** it. For a non-static method, you need an object of the class to call it using the dot (.) operator. For a static method, you can call it directly using the class name.

* **Methods with a return value:** You can assign the returned value to a variable (e.g., int result = minFunction(a, b);).
* **Methods with void:** They don't return a value. The call is a simple statement (e.g., methodRankPoints(255.7);).

**2. Passing Parameters by Value**

In Java, arguments are always passed by **value**. This means that a copy of the argument's value is passed to the method's parameter. Any changes made to the parameter inside the method do **not** affect the original argument's value outside the method.

**3. Method Overloading**

**Method overloading** is a form of polymorphism where a class has two or more methods with the **same name** but a **different number or type of parameters**. This makes code more readable and flexible. The Java compiler determines which method to call based on the arguments provided during the method call.

**4. The this Keyword**

The this keyword is a reference to the **current object**. It's primarily used within a method or constructor to:

* **Distinguish** between an instance variable and a local variable with the same name.
* **Invoke** another constructor within the same class (this()).

**5. Variable Arguments (Var-args)**

Introduced in JDK 1.5, **var-args** allow a method to accept a variable number of arguments of the same type. The parameter is declared with an ellipsis (...) after the data type (e.g., double... numbers). The method treats this parameter as an array. A method can have only one var-args parameter, and it must be the last one in the list.

**6. Command-Line Arguments**

The main() method's String[] args parameter is used to accept **command-line arguments** as an array of strings. This allows you to pass information to your program when it's executed from the command line.

**7. The finalize() Method**

The protected void finalize() method is called by the garbage collector just before an object is destroyed. It can be used for cleanup tasks. However, its execution is **not guaranteed**, so it should not be relied upon for critical resource management.

**Java - Variable Scopes**

The scope of a variable defines where it can be created and accessed within a program, as well as its lifetime. There are three main types of variable scopes in Java.

**1. Instance Variable Scope 🏠**

**Instance variables** are declared inside a class but outside of any method or block. Their scope extends throughout the entire class, meaning they can be accessed by any of the class's non-static methods. The lifetime of an instance variable is tied to the object it belongs to; it exists as long as the object remains in memory.

* **Example:** In the provided text, puppyAge is an instance variable. It's available to methods like setAge() and getAge() but would not be accessible from a static method without an object.

**2. Class (Static) Variable Scope 🌍**

A **class variable** is declared inside a class, outside of any block, and is marked with the **static keyword**. Its scope is the entire class, and its lifetime lasts for the duration of the program, or as long as the class is loaded into memory. Static variables can be accessed directly using the class name, without needing an object.

* **Example:** The BREED variable is a static class variable. It can be accessed directly via Puppy.BREED from anywhere within the class or even from other classes.

**3. Local Variable Scope 📦**

**Local variables** are declared inside a method, constructor, or a block. Their scope is strictly limited to the block in which they are declared. They are created when the block is entered and are destroyed once the control flow leaves that block.

* **Example:** In the main() method, variables like a, b, and c are local variables. They can only be accessed and used within the main method's curly braces {}.

**Key Takeaways ✨**

* **Instance variables** belong to an object; their scope is the class, and they live as long as the object does.
* **Static variables** belong to the class itself; their scope is the whole class, and they live as long as the program runs.
* **Local variables** belong to a block of code; their scope is limited to that block, and they are destroyed when the block is exited.

Java also uses **access modifiers** (public, private, protected, and default) to control the visibility of variables, which works in conjunction with scope to determine where they can be accessed.

**CONSTRUCTORS IN JAVA :**

A **Java constructor** is a unique type of method that's automatically called when a new object is created. Its main purpose is to **initialize the new object's state**, ensuring all its instance variables have valid starting values before the object is used. While it looks like a method, a constructor has no explicit return type, not even void, because it implicitly returns a newly created instance of its class.

**The Importance of Constructors**

Constructors are essential for two main reasons:

* **Initialization:** They provide the first opportunity to assign values to an object's instance variables. This is crucial for making sure the object is in a valid and usable state from the moment it is created. Without a constructor, you'd have to create an object and then call separate setter methods to set its values, which is less efficient and more error-prone.
* **Enforcing Rules:** By defining a constructor, you can enforce rules for object creation. For example, if a Car object must have a brand and a model to be valid, you can create a parameterized constructor that requires these values. This prevents the creation of "empty" or incomplete objects.

**Types of Constructors**

There are three key types of constructors you'll encounter in Java.

A diagram of a construction site

AI-generated content may be incorrect.

**Default Constructor**

This is a **compiler-generated** constructor. If you don't write any constructors for your class, the Java compiler automatically adds a public, no-argument constructor behind the scenes. Its sole job is to initialize all instance variables to their default values (e.g., 0 for numeric types, null for objects, false for booleans). The moment you write even one constructor yourself, the compiler's default constructor is no longer provided.

**No-Args Constructor**

This is a **user-defined** constructor that takes **no parameters**. You would create this type of constructor to perform specific setup procedures or to initialize instance variables to a non-default, pre-determined value. It's often used to provide a simple way to create a basic instance of a class.

**Parameterized Constructor**

This is a **user-defined** constructor that takes **one or more parameters**. This is the most common type of constructor, as it allows you to create a fully customized object by passing in its initial state at the time of creation. This is much more convenient and readable than creating a blank object and then calling a series of setter methods to populate it.

**Constructor Overloading**

**Constructor overloading** is the ability to have multiple constructors within a single class. Each constructor must have a unique **parameter list** (either a different number of parameters or different parameter types). This provides flexibility for the user of your class, allowing them to create objects in different ways. For example, a Person class might have constructors for creating a person with just a name, or one with a name and an age.

**The this Keyword in Constructors**

The **this** keyword is a powerful tool within a constructor. It serves two main purposes:

1. **Resolving Ambiguity:** It's used to distinguish between an instance variable and a constructor parameter when they have the same name. For example, in Person(String name), this.name = name; correctly assigns the value from the parameter to the instance variable.
2. **Chaining Constructors:** You can use this() to call one constructor from another within the same class. This is known as **explicit constructor invocation** and is useful for reusing common initialization code. For example, a parameterized constructor might call a no-args constructor to perform some initial setup before its own specific initialization.

**ACCESS MODIFIERS :**

**Java Access Modifiers**

Java access modifiers are keywords that set the **visibility and scope** of classes, methods, variables, and constructors. They're a key part of encapsulation, helping to control what parts of your code can be accessed by other classes. There are four types of access modifiers: default, private, protected, and public.

**public 🌐**

The public modifier is the least restrictive. A public class, method, or variable can be accessed from **any other class** in the Java universe, regardless of the package. This is why the main() method is always public—it needs to be accessible to the Java Virtual Machine (JVM) to run your program.

* **Rule:** Can be applied to a class, method, or variable.
* **Access:** Accessible from anywhere.

**protected 🛡️**

The protected modifier provides a middle ground. A protected member can be accessed by any class within the **same package**, as well as by any **subclass**, even if the subclass is in a different package. This is ideal for methods or variables that are intended for inheritance but not for general use.

* **Rule:** Cannot be applied to a class. Can be applied to a method or variable.
* **Access:** Within the same package and by all subclasses.

**default (Package-Private) 📦**

The default access modifier is used when no explicit modifier is specified. It's also known as **package-private**. A default-access member is visible only to other classes within the **same package**. It's useful for classes or members that serve as internal helpers for a specific package and aren't meant for external use.

* **Rule:** Can be applied to a class, method, or variable.
* **Access:** Only within the same package.

**private 🤫**

The private modifier is the most restrictive. A private member can **only be accessed within the declared class itself**. This is the core mechanism of encapsulation. By making instance variables private and providing public getter and setter methods, you control how the data can be accessed and modified, protecting it from invalid changes.

* **Rule:** Cannot be applied to a class or an interface. Can be applied to a method or a variable.
* **Access:** Only within the declared class.

A table with black text

AI-generated content may be incorrect.

**INHERITANCE :**

Inheritance is a fundamental pillar of object-oriented programming (OOP) in Java. It allows one class to acquire the properties and behaviors (methods and attributes) of another, creating a hierarchical "IS-A" relationship. This promotes **code reusability** and **extensibility**.

**Core Concepts and Terminology**

* **Subclass (Child Class):** A class that inherits properties from another class.
* **Superclass (Parent Class):** The class whose members are inherited.
* **extends Keyword:** The keyword used to create an inheritance relationship. class Dog extends Animal means Dog inherits from Animal.
* **Inherited Members:** A subclass inherits all public and protected members of its superclass. **Private members and constructors are not inherited.**

**The super Keyword**

The super keyword is used within a subclass to refer to members of its immediate superclass. It's crucial for two main tasks:

1. **Invoking the Superclass Constructor:** A subclass must call a superclass's constructor. If you don't explicitly call super(), the compiler automatically inserts a call to the no-argument superclass constructor. Use super(arguments) to call a parameterized superclass constructor.
2. **Differentiating Members:** If a subclass has a member with the same name as a superclass member, super.member is used to access the superclass's version, preventing it from being hidden.

**Relationships: IS-A vs. HAS-A**

* **IS-A (Inheritance):** A relationship of specialization. A Car **IS-A** Vehicle. This is a tight coupling, and it means the subclass can be treated as an object of its superclass. The instanceof operator can verify this relationship at runtime.
* **HAS-A (Composition):** A relationship of containment. A Car **HAS-A** an Engine. This is implemented by making an instance of one class an attribute within another. This approach promotes code reuse and a more flexible design.

**Method Overriding**

Inheritance is a prerequisite for **method overriding**, a key feature of polymorphism. Method overriding is when a subclass provides its own specific implementation for a method that is already defined in its superclass. The method signatures (name, return type, and parameters) must be identical. Overriding allows a subclass to provide a unique behavior while adhering to the superclass's structure.

**Types of Inheritance in Java 🌳**

Java supports three types of inheritance using the extends keyword. It's important to note that Java **does not support multiple or hybrid inheritance** to avoid the complexities of the **"Diamond Problem."**

A diagram of a class

AI-generated content may be incorrect.

Java uses **interfaces** to solve the need for multiple inheritance. A class can implement multiple interfaces, which contain method declarations without implementation, effectively allowing a class to adopt multiple behaviors without the ambiguity of multiple inheritance.

**AGGREGATION :**

Aggregation in Java is a specific type of association between two classes where one class **"has a"** relationship with another. It means that an object of one class contains a reference to an object of another class. This relationship is considered "weak" because the contained object can exist independently of the containing object.

**Key Concepts**

* **HAS-A Relationship:** Aggregation models this relationship. For example, a Student has an Address.
* **Code Reusability:** It allows for code reuse because one class can utilize the methods and attributes of the object it contains, without needing to inherit from it.
* **Independent Existence:** The key characteristic of aggregation is that the "contained" object can exist on its own. For instance, an Address object can exist independently of the Student object it's associated with. If the Student object is destroyed, the Address object remains. This is a crucial distinction from **composition**, where the contained object cannot exist without its parent.

**SUPER KEYWORD :**

The super keyword in Java is a reference variable used to access members (methods, constructors, and variables) of the immediate **superclass** from within a subclass. It's a key tool for managing inheritance.

**1. Invoking a Superclass Constructor**

A subclass constructor implicitly calls the no-argument constructor of its superclass. Use super(arguments) to explicitly call a specific parameterized constructor of the superclass. This call must be the **first statement** in the subclass's constructor.

**Example:**

Java

class Animal {

Animal(String name) {

System.out.println("Animal constructor called with: " + name);

}

}

class Dog extends Animal {

Dog() {

super("Buddy"); // Calls the Animal constructor

System.out.println("Dog constructor called.");

}

public static void main(String[] args) {

new Dog();

}

}

**Output:**

Animal constructor called with: Buddy

Dog constructor called.

**2. Accessing a Superclass Method**

When a subclass **overrides** a method from its superclass, you can use super.methodName() to call the overridden version. This is useful for extending the superclass's functionality without completely replacing it.

**Example:**

Java

class Vehicle {

void run() {

System.out.println("Vehicle is running.");

}

}

class Car extends Vehicle {

@Override

void run() {

super.run(); // Calls the superclass's run() method

System.out.println("Car is running on the road.");

}

public static void main(String[] args) {

new Car().run();

}

}

**Output:**

Vehicle is running.

Car is running on the road.

**3. Accessing a Superclass Variable**

If a subclass declares a variable with the same name as a variable in its superclass, the subclass's variable **hides** the superclass's variable. You use super.variableName to access the hidden superclass variable.

**Example:**

Java

class Parent {

int num = 10;

}

class Child extends Parent {

int num = 20; // Hides the superclass's num

void display() {

System.out.println("Subclass num: " + num);

System.out.println("Superclass num: " + super.num);

}

public static void main(String[] args) {

new Child().display();

}

}

**Output:**

Subclass num: 20

Superclass num: 10