**ENCAPSULATION**

It is defined as the **wrapping up of data under a single unit.** It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

* Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.
* In encapsulation, the data in a class is hidden from other classes, which is similar to what **data-hiding** does. So, the terms “encapsulation” and “data-hiding” are used interchangeably.
* Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

class BankAccount {

private double balance;

public void deposit(double amount) {

if (amount > 0)

balance += amount;

}

public double getBalance() {

return balance;

}

}

public class EncapsulationExample {

public static void main(String[] args) {

BankAccount account = new BankAccount();

account.deposit(5000);

System.out.println("Balance: " + account.getBalance());

}

}

**INHERITANCE**

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using extends keyword. Inheritance is also known as “is-a” relationship.

Let us discuss some frequently used important terminologies:

* Superclass: The class whose features are inherited is known as superclass (also known as base or parent class).
* Subclass: The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
* Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

class Vehicle {

public void startEngine() {

System.out.println("Engine started");

}

}

class Car extends Vehicle {

public void drive() {

System.out.println("Car is driving");

}

}

public class InheritanceExample {

public static void main(String[] args) {

Car myCar = new Car();

myCar.startEngine(); // inherited method

myCar.drive();

}

}

**POLYMORPHISM**

It refers to the ability of object-oriented programming languages t**o differentiate between entities with the same name efficiently**. This is done by Java with the help of the signature and declaration of these entities. The ability to appear in many forms is called [polymorphism.](https://www.geeksforgeeks.org/polymorphism-in-java/)

**Example:**

sleep(1000) *//millis*

sleep(1000,2000) *//millis,nanos*

**Types of Polymorphism**

Polymorphism in Java is mainly of 2 types as mentioned below:

1. [Method Overloading](https://www.geeksforgeeks.org/overloading-in-java)
2. [Method Overriding](https://www.geeksforgeeks.org/overriding-in-java)

**Method Overloading and Method Overriding**

**1. Method Overloading:** Also, known as **compile-time polymorphism**, is the concept of Polymorphism where more than one method share the same name with different signature(Parameters) in a class. The return type of these methods can or cannot be same.

**2. Method Overriding:**Also, known as **run-time polymorphism**,is the concept of Polymorphism where method in the child class has the same name, return-type and parameters as in parent class. The child class provides the implementation in the method already written.

**Below is the implementation of both the concepts:**

*// Method Overloading and Overriding*

*// Parent Class*

**class** **Parent** {

*// Overloaded method (compile-time polymorphism)*

**public** void func() {

System.out.println("Parent.func()");

}

*// Overloaded method (same name, different parameter)*

**public** void func(int a) {

System.out.println("Parent.func(int): " + a);

}

}

*// Child Class*

**class** **Child** **extends** Parent {

*// Overrides Parent.func(int) (runtime polymorphism)*

@Override

**public** void func(int a) {

System.out.println("Child.func(int): " + a);

}

}

**public** **class** **Main** {

**public** **static** void main(String[] args) {

Parent parent = **new** Parent();

Child child = **new** Child();

*// Dynamic dispatch*

Parent polymorphicObj = **new** Child();

*// Method Overloading (compile-time)*

parent.func();

parent.func(10);

*// Method Overriding (runtime)*

child.func(20);

*// Polymorphism in action*

polymorphicObj.func(30);

}

}

**Output**

Parent.func()

Parent.func(int): 10

Child.func(int): 20

Child.func(int): 30

**ABSTRACTION**

**Data Abstraction** is the property by virtue of which **only the essential details are displayed to the user.** The trivial or non-essential units are not displayed to the user. Data Abstraction may also be defined as the process of identifying only the required characteristics of an object, ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the object.

**Real-life Example:**Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the car speed or applying brakes will stop the car, but he does not know how on pressing the accelerator, the speed is actually increasing. He does not know about the inner mechanism of the car or the implementation of the accelerators, brakes etc. in the car. This is what abstraction is.

**Note:** In Java, abstraction is achieved by [interfaces](https://www.geeksforgeeks.org/interfaces-in-java) and [abstract classes](https://www.geeksforgeeks.org/abstract-classes-in-java). We can achieve 100% abstraction using interfaces.

abstract class Shape {

abstract void draw(); // abstract method

}

class Circle extends Shape {

public void draw() {

System.out.println("Drawing Circle");

}

}

public class AbstractionExample {

public static void main(String[] args) {

Shape s = new Circle();

s.draw();

}

}

**final (Keyword)**

**📚 Meaning:**

* final is a **keyword** used to apply restrictions on classes, methods, and variables.

**🔥 Usage:**

* **Final Variable** ➔ Value **cannot be changed** (constant).
* **Final Method** ➔ Method **cannot be overridden**.
* **Final Class** ➔ Class **cannot be inherited**.

**💻 Java Example:**

final int speedLimit = 80; // Final variable

// speedLimit = 100; // ❌ Error: can't assign new value

final class Car { // Final class

final void start() { // Final method

System.out.println("Car starts");

}

}

// class BMW extends Car {} // ❌ Error: cannot extend final class

**finally (Block)**

**📚 Meaning:**

* finally is a **block of code** that **always executes** after a try-catch, whether an exception occurs or not.
* Used mainly for **cleanup activities** like closing a file, releasing database connection, etc.

**💻 Java Example:**

public class FinallyExample {

public static void main(String[] args) {

try {

int data = 50 / 0; // This will throw ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Exception caught: " + e);

} finally {

System.out.println("Finally block is always executed");

}

}

}

✅ Output:

Exception caught: java.lang.ArithmeticException: / by zero

Finally block is always executed

**finalize() (Method)**

**📚 Meaning:**

* finalize() is a **method** defined in Object class.
* It is **called by Garbage Collector** before destroying an object.
* Purpose: To **perform cleanup** activities before object is removed from memory.

**💻 Java Example:**

public class FinalizeExample {

public void finalize() {

System.out.println("Finalize method called");

}

public static void main(String[] args) {

FinalizeExample obj = new FinalizeExample();

obj = null; // Eligible for Garbage Collection

System.gc(); // Request JVM to call Garbage Collector

}

}

✅ Output (may vary):

Finalize method called

*(Note: System.gc() is just a request. GC timing is decided by JVM.)*

**🆚 Quick Summary Table:**

| **Keyword/Method** | **Purpose** | **Where used** |
| --- | --- | --- |
| final | To **restrict modification** | With variables, methods, classes |
| finally | To **always execute** a block | With try-catch blocks (Exception Handling) |
| finalize() | To **perform cleanup before destruction** | When object is garbage collected |
| **⚡ Important Points:**   * finalize() **is protected** in Object class, so when you override, it should be protected (or broader visibility like public). * It **is not guaranteed** exactly *when* finalize() will be called — it depends on the JVM's Garbage Collector. * After Java 9, **finalize() is deprecated** because it's considered unreliable and can cause performance/memory problems.   **🚨 Important in interviews:**   * **Yes, we can override finalize().** * **But it is not recommended to use finalize() in modern Java applications.** * Instead, use **try-with-resources** or **explicit resource management**.   **🎯 In Short:**   | **Question** | **Answer** | | --- | --- | | Can finalize() be overridden? | ✅ Yes | | Is it a final method? | ❌ No | | Should we rely on finalize()? | ⚠️ Not recommended (deprecated) |   **✍ Small Meaning of Each State:**   | **State** | **When?** | **Example Methods** | | --- | --- | --- | | New | When you create a thread object but not started it yet | Thread t = new Thread(); | | Runnable | After calling start(), thread ready to run | t.start(); | | Running | JVM CPU scheduler picks the thread | Automatically by CPU | | Blocked | Waiting to acquire lock | When using synchronized block | | Waiting | Waiting forever for other thread | join(), wait() | | Timed Waiting | Waiting for a certain time | sleep(1000), wait(1000), join(5000) | | Terminated | After run() method ends or exception happens | After completion | |  |  |

| **Feature** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| --- | --- | --- | --- |
| **Order** | Unordered | Maintains insertion order | Sorted (natural order or custom comparator) |
| **Performance** | Best performance (O(1) average) | Slightly slower (due to linked list) | Slower (O(log n) due to sorting) |
| **Duplicates** | No duplicates | No duplicates | No duplicates |
| **Null Values** | Allows one null element | Allows one null element | Does not allow null elements |
| **Example** | Phone number lookup | Music playlist (order matters) | Game leaderboard (ranking by score) |

| **Feature** | **PriorityQueue** | **Deque** |
| --- | --- | --- |
| **Purpose** | Process elements based on priority (not insertion order) | Insert/remove elements from both ends |
| **Order** | Sorted based on natural ordering or comparator | Insertion order (FIFO or LIFO) |
| **Usage Example** | Hospital ER (highest priority patient first) | Card deck (add/remove from both ends) |
| **Methods** | add(), poll(), peek() | addFirst(), addLast(), removeFirst(), removeLast() |

| **Feature** | **ArrayList** | **LinkedList** | **Vector** |
| --- | --- | --- | --- |
| **Underlying Structure** | Dynamic array | Doubly linked list | Dynamic array |
| **Order** | Maintains insertion order | Maintains insertion order | Maintains insertion order |
| **Performance** | Fast access, slower insertions | Slow access, fast insertions/removals | Slower than ArrayList (due to synchronization) |
| **Thread-Safety** | Not thread-safe | Not thread-safe | Thread-safe (methods are synchronized) |
| **Growth Factor** | Increases by 50% when capacity is exceeded | Grows dynamically based on elements added | Doubles its size when capacity is exceeded |
| **Use Case** | General-purpose list with fast access | Frequent insertions/deletions, particularly at ends | Thread-safe applications, less common today |
|  |  |  |  |

| **Feature** | **HashMap** | **LinkedHashMap** | **TreeMap** |
| --- | --- | --- | --- |
| **Underlying Structure** | Hash table | Hash table + doubly linked list | Red-Black tree (balanced binary search tree) |
| **Order** | Unordered | Maintains insertion order | Sorted by natural order or custom comparator |
| **Performance** | Fast (O(1) for most operations) | Slightly slower due to linked list | Slower (O(log n) due to sorting) |
| **Thread-Safety** | Not thread-safe | Not thread-safe | Not thread-safe |
| **Null Keys and Values** | Allows 1 null key and multiple null values | Allows 1 null key and multiple null values | No null keys, but allows null values |
| **Use Case** | General-purpose map (lookup, insert) | Maintain insertion order while using a map | Sorted maps (ranking, leaderboard) |