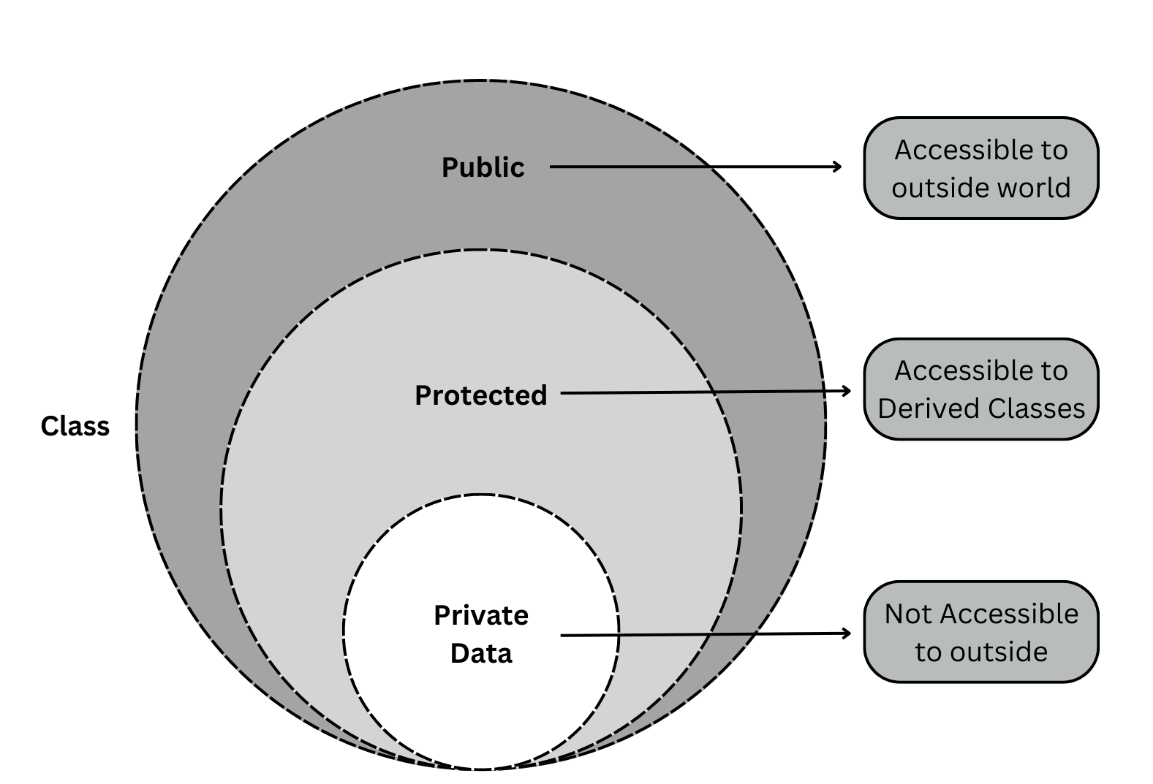
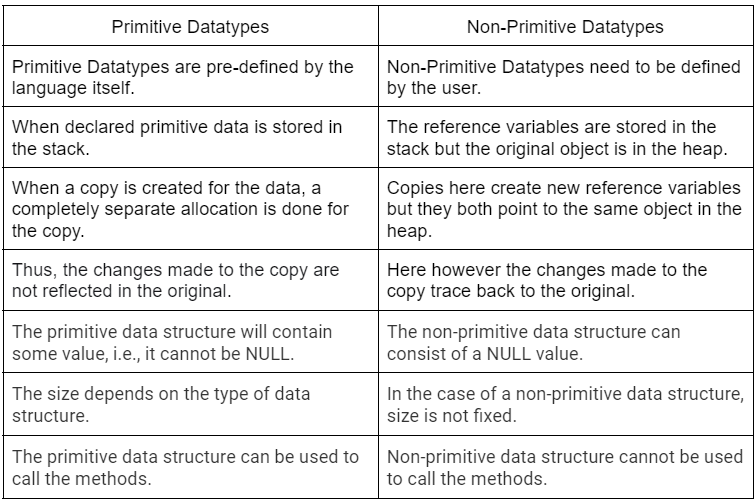
# Public , Private , Protected



https://chatgpt.com/share/0e412a79-2cc2-44ed-97cf-90ed7bf8bb4e/

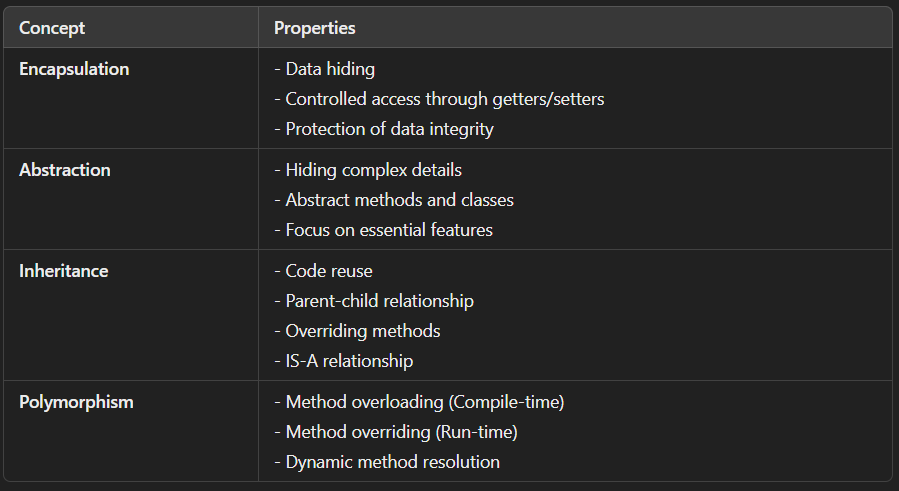
# Data Types



# **Classes:**

A class in C++ is like a blueprint for creating objects. It defines the characteristics (data members) and behaviors (member functions) that the objects will have

# Core OOP:



# **Encapsulation:**

Data encapsulation is the process of combining data and the functions or methods that operate on that data into a unique unit that is protected from outside interference and misuse.

It helps in protecting the data from being directly accessed from outside the class.

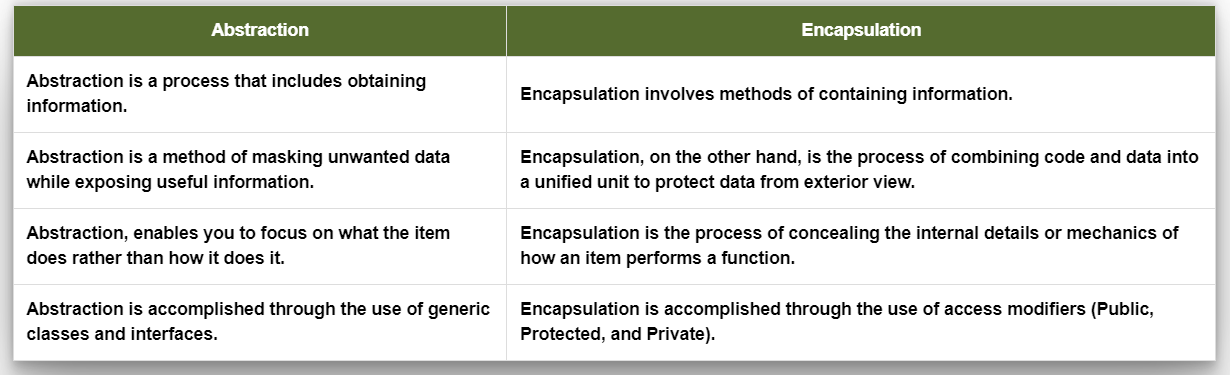
## **Why Use Encapsulation?**

* **Data Hiding:** Protects the internal state of an object from unintended interference.
* **Modularity:** Keeps the code organized and manageable.
* **Maintainability:** Makes it easier to change the implementation without affecting other parts of the program.

|  |
| --- |
| class Employee {  private String name; // private variable  // Getter method to access the private variable  public String getName() {  return name;  }  // Setter method to modify the private variable  public void setName(String name) {  this.name = name;  }  }  public class Main {  public static void main(String[] args) {  Employee emp = new Employee();  emp.setName("Alice");  System.out.println(emp.getName()); // Output: Alice  }  } |

# **Abstraction:**

Abstraction is the process of hiding complex implementation details and showing only the necessary features. Think of it as using a car; you don't need to know how the engine works to drive it.



|  |
| --- |
| abstract class Animal {  // Abstract method (no body)  abstract void sound();  // Concrete method  void sleep() {  System.out.println("Sleeping...");  }  }  class Dog extends Animal {  void sound() {  System.out.println("Woof Woof");  }  }  public class Main {  public static void main(String[] args) {  Dog dog = new Dog();  dog.sound(); // Output: Woof Woof  dog.sleep(); // Output: Sleeping...  }  } |

# **Inheritance**

Inheritance allows one class (child class) to inherit fields and methods from another class (parent class). It promotes code reusability

|  |
| --- |
| class Animal {  void eat() {  System.out.println("Eating...");  }  }  class Dog extends Animal {  void bark() {  System.out.println("Barking...");  }  }  public class Main {  public static void main(String[] args) {  Dog dog = new Dog();  dog.eat(); // Output: Eating... (Inherited from Animal)  dog.bark(); // Output: Barking...  }  } |

# **Polymorphism**

Polymorphism allows one method to have different behaviors based on the object calling it. It is achieved through method overloading or method overriding.

**Types of Polymorphism:**

1. **Compile-time (Method Overloading):** Same method name but different parameters.
2. **Run-time (Method Overriding):** Same method name but behavior depends on the object (child class or parent class).

|  |
| --- |
| class Calculator {  // Overloaded methods  int add(int a, int b) {  return a + b;  }  int add(int a, int b, int c) {  return a + b + c;  }  }  public class Main {  public static void main(String[] args) {  Calculator calc = new Calculator();  System.out.println(calc.add(5, 10)); // Output: 15  System.out.println(calc.add(5, 10, 20)); // Output: 35  }  } |

|  |
| --- |
| class Animal {  void sound() {  System.out.println("Animal sound");  }  }  class Dog extends Animal {  @Override  void sound() {  System.out.println("Woof Woof");  }  }  public class Main {  public static void main(String[] args) {  Animal animal = new Dog(); // Polymorphism  animal.sound(); // Output: Woof Woof (Dog's version of sound is called)  }  } |

# **Interfaces**

An **interface** in Java is a reference type, similar to a class, that can contain **abstract methods** (methods without a body) and **constants** (fields that are implicitly public, static, and final). It is a way to define a contract that any implementing class must follow, ensuring that a class will provide specific methods without dictating how these methods should be implemented.

**When to Use Interfaces:**

* **Multiple Inheritance**: In Java, a class cannot inherit from more than one class, but it can implement multiple interfaces, enabling a form of multiple inheritance.
* **Loose Coupling**: Interfaces allow you to decouple the implementation from the code that uses the interface, making your code more flexible and easier to maintain.
* **Common Behavior Across Unrelated Classes**: If several unrelated classes need to share certain behaviors, you can define an interface for that behavior and implement it in all those classes.

|  |
| --- |
| // Defining the interface  interface Vehicle {  // Abstract methods (no body)  void start();  void stop();  int getSpeed();  }  // Implementing the interface in the Car class  class Car implements Vehicle {  private int speed;  // Constructor for Car  public Car(int speed) {  this.speed = speed;  }  // Implement the start method  @Override  public void start() {  System.out.println("Car is starting.");  }  // Implement the stop method  @Override  public void stop() {  System.out.println("Car is stopping.");  }  // Implement the getSpeed method  @Override  public int getSpeed() {  return speed;  }  // Car-specific method  public void openTrunk() {  System.out.println("Car trunk is open.");  }  }  // Implementing the interface in the Bike class  class Bike implements Vehicle {  private int speed;  // Constructor for Bike  public Bike(int speed) {  this.speed = speed;  }  // Implement the start method  @Override  public void start() {  System.out.println("Bike is starting.");  }  // Implement the stop method  @Override  public void stop() {  System.out.println("Bike is stopping.");  }  // Implement the getSpeed method  @Override  public int getSpeed() {  return speed;  }  // Bike-specific method  public void popWheelie() {  System.out.println("Bike is popping a wheelie.");  }  }  // Main class to test the interface  public class Main {  public static void main(String[] args) {  // Create an object of Car  Vehicle myCar = new Car(120);  myCar.start();  System.out.println("Car speed: " + myCar.getSpeed());  myCar.stop();  // Create an object of Bike  Vehicle myBike = new Bike(80);  myBike.start();  System.out.println("Bike speed: " + myBike.getSpeed());  myBike.stop();  }  } |