Encapsulation is a fundamental concept in object-oriented programming (OOP) that bundles data (variables) and the methods that operate on that data into a single unit, known as a **class**. It hides the internal state of an object from the outside world and only allows access to it through a controlled interface.

Think of it like a pill capsule: the capsule itself is the class, the medicine inside is the data, and you can only take the medicine by swallowing the capsule, which is the controlled method of access. The internal workings of the medicine are hidden from you, but you can still use it for its intended purpose.

**Key Principles of Encapsulation**

* **Data Hiding:** This is the core of encapsulation. You declare class variables as **private** so they can't be accessed directly from outside the class. This prevents unintended modifications.
* **Controlled Access:** You provide **public** methods, known as **getters** and **setters**, to read and modify the private variables.
  + **Getters** are used to retrieve the value of a variable.
  + **Setters** are used to update the value of a variable, often including validation logic to ensure the new value is valid.

**Example**

Let's create a Student class to demonstrate encapsulation. Without encapsulation, you might have a Student class with a private variable age and a public setter method setAge that validates the age. You could also have a private variable name and a public getter method getName.

Here is the code with encapsulation:

Java

public class Student {

// Private instance variables (data hiding)

private String name;

private int age;

// Public constructor

public Student(String name, int age) {

this.name = name;

// Use the setter to enforce validation during object creation

setAge(age);

}

// Public getter method for name

public String getName() {

return name;

}

// Public setter method for name

public void setName(String name) {

this.name = name;

}

// Public getter method for age

public int getAge() {

return age;

}

// Public setter method with validation (controlled access)

public void setAge(int age) {

if (age > 0) { // Validation check

this.age = age;

} else {

System.out.println("Invalid age. Age must be a positive number.");

}

}

}

Now let's see how you would use this Student class in a main method.

Java

public class Main {

public static void main(String[] args) {

// Create a new Student object

Student student1 = new Student("Alice", 20);

// Access data using public getter methods

System.out.println("Student Name: " + student1.getName()); // Output: Student Name: Alice

System.out.println("Student Age: " + student1.getAge()); // Output: Student Age: 20

// Attempt to modify data using public setter methods

student1.setAge(25);

System.out.println("New Age: " + student1.getAge()); // Output: New Age: 25

// Attempt to set an invalid age

student1.setAge(-5); // Output: Invalid age. Age must be a positive number.

System.out.println("Age after invalid attempt: " + student1.getAge()); // Output: Age after invalid attempt: 25

}

}

In this example, the age variable is private, so you can't access it directly like student1.age = -5;. Instead, you must go through the setAge() method, which performs a check to ensure the age is valid. This prevents invalid data from being stored in the object, making the code more robust and predictable.

**Benefits of Encapsulation**

* **Increased Flexibility:** You can change the internal implementation of a class without affecting the external code that uses it. For example, you could change the way age is stored (e.g., from an int to a Date of birth) without breaking the code that uses the getAge() and setAge() methods.
* **Data Integrity:** It ensures that an object's state is always valid by preventing direct, uncontrolled access to its data.
* **Reduced Complexity:** It simplifies the user's interaction with the object by providing a clear, simple interface. The user doesn't need to know how the data is stored or manipulated internally.
* **Better Maintainability:** It makes code easier to maintain and debug because changes are localized within the class.