Encapsulation in Object-Oriented Programming

What is Encapsulation?

Encapsulation is one of the four fundamental OOP principles. It refers to **bundling data and methods** that operate on that data within a single unit (class), restricting direct access to some of the object's components.

The four fundamental Object-Oriented Programming (OOP)

- **Encapsulation:** Bundling data and methods that operate on that data within a class, restricting direct access to some components.
- **Abstraction:** Hiding complex implementation details and exposing only the necessary features of an object.
- **Inheritance:** Allowing a class to inherit properties and behaviors (methods) from another class.
- **Polymorphism:** Enabling objects to be treated as instances of their parent class, allowing for method overriding and dynamic method invocation.

Motivation example

```
String[] names = {"Alice", "Bob", "Charlie"};
String[] surnames = {"Smith", "Johnson", "Williams"};

for (int i = 0; i < names.length; i++) {
    System.out.println(names[i] + " " + surnames[i]);
}
```

What's wrong with this code?

Problem!

Arrays are independent and can be accessed or modified separately.

Related data is not grouped together.

Solution: Encapsulation

```
public class Person {
    public String name;
    public String surname;

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
}
```

The class encapsulates related data (name and surname) into a single unit (Person). The constructor helps initialize right objects.

Solution: Encapsulation (II)

```
Person[] people = {
    new Person("Alice", "Smith"),
    new Person("Bob", "Johnson"),
    new Person("Charlie", "Williams")
};

for (Person person : people) {
    System.out.println(person.name + " " + person.surname);
}
```

We can do it better?

Solution: Encapsulation (III)

```
public class Person {
    public String name;
    public String surname;

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    // Method to get full name
    public String getFullName() {
        return name + " " + surname;
    }
}
```

The code for printing the full name is now encapsulated within the `Person` class.

Solution: Encapsulation (IV)

```
Person[] people = {
    new Person("Alice", "Smith"),
    new Person("Bob", "Johnson"),
    new Person("Charlie", "Williams")
};

for (Person person : people) {
    System.out.println(person.getFullName());
}
```

Now, the main code is cleaner and more maintainable.

What is a class?

So then, classes are:

- **Data**
- **Code that operates on that data**

Non OOP programmers have done "this" manually, putting data and related functions in the same file.

OOP languages provide syntax to do this easily.

Why Encapsulation Matters

- Protects internal state
- Prevents unintended interference
- Simplifies code maintenance
- Enables modular design

Key Principles of Encapsulation

- Hide internal details
- Expose only necessary functionality
- Control access via methods

Why hiding details?

```
class BankAccount {
    public double balance;
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
}
```

Any code can modify `balance` directly, leading to potential inconsistencies.

```
BankAccount account = new BankAccount(100);
account.balance = account.balance - 200; // Invalid state
```

Why hiding details? (II)

```
class BankAccount {
    private double balance;
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
    public double getBalance() {
        return balance;
    }
    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Insufficient funds");
        }
    }
}
```

Now we cannot modifiy `balance` directly. And we can add checks in `withdraw` method.

Why hiding details? (III)

Imagine that Elon Musk decides to open an account in our bank. `double` is not big enough to hold his balance :)

What would happen if we want to change the internal representation of `balance` to a `BigDecimal` instead of a `double`?

Why hiding details? (IV)

```
class BankAccount {
    private BigDecimal balance;

    public BankAccount(BigDecimal initialBalance) {
        this.balance = initialBalance;
    }

    public BankAccount(double initialBalance) { //Keep old constructor for compatibility
        this.balance = BigDecimal.valueOf(initialBalance);
    }

    public double getBalance() // old code still works {
        if(balance>Double.MAX_VALUE) {
            throw new ArithmeticException("Balance too large to fit in a double");
        }
        return balance.doubleValue();
    }

    public BigDecimal getBigBalance() { // new code for Elon
        return balance;
    }
}
```

Encapsulation in Java

Java uses classes to encapsulate data and behavior. Access modifiers (`private` , `protected` , `public`) control visibility.

Access Modifiers in Java

- `private` : Only accessible within the class
- `protected` : Accessible within package and subclasses
- `public` : Accessible from anywhere

Example: Private Fields

```
public class Person {
    private String name;
    private int age;
}
```

Fields are hidden from outside code.

Example: Getters and Setters

```
public class Person {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

- We can add validation in setters if needed.
- We can hide internal representation in getters.
- Intellij can generate these for you!

Benefits of Encapsulation

- Improved security
- Easier refactoring
- Clear API boundaries
- Reduced complexity

Encapsulation vs. Abstraction

- **Encapsulation**: Hides internal state and implementation
- **Abstraction**: Hides complexity by exposing only relevant features

Common Mistakes

- Exposing fields as `public`
- Not using setters/getters
- Breaking encapsulation for convenience

Best Practices

- Always use `private` for fields
- Provide controlled access via methods
- Validate data in setters

Summary

Encapsulation is essential for robust, maintainable, and secure object-oriented code. Use access modifiers and methods to protect your data.