Welcome back to the

Java Course

Module 3 - OOP

Encapsulation

In Object-Oriented Programming, one of the fundamental concepts is **encapsulation**.

It involves restricting and managing the visibility of the contents of an object.

Encapsulation

We've already seen one aspect of encapsulation, which is setting attributes as private.

This prevents interacting with them outside of the class.

Encapsulation

Another aspect is data protection.

Encapsulation protects the data of an object by using '*get*' and '*set*' methods as intermediaries.

Encapsulation

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

Encapsulation

Suppose we need to modify a class that is frequently used in a complex software.

Encapsulation

Without encapsulation, we would also need to change how we use the class throughout the software, even if we use it dozens or hundreds of times!

Encapsulation

With encapsulation, on the other hand, we interact with the class solely through the use of access methods.

This allows us to modify only the class itself without affecting the rest of the code.

Encapsulation

This way maintenance, updates, and debugging of the code become much simpler and more flexible.

Example

Let’s see an example of encapsulation.

Encapsulation

Let's assume we want to modify a bank balance.

For security reasons, before being able to modify it, certain checks must be passed.

Encapsulation

Login, password, entering a debit or credit card ID, and demonstrating physical presence are some of the necessary checks.

Encapsulation

The 'set' method allows modifying the balance value only if all the checks are successful.

But without encapsulation, anyone would have direct access to the balance!

Encapsulation

this.balance = 1,000,000,000;

*I am rich !!! ^^*

Accessibility

Public :

Access is allowed within the entire folder in which the class is defined.

Accessibility

Private :

Access is allowed only within the class in which the attribute or method is defined.

Accessibility

Protected :

Access is allowed only within the class in which the attribute or method is defined and its subclasses.

Let’s practice!

Exceptions

When we create code, we sometimes have to handle situations that are impossible for the compiler to execute.

In these cases, the execution is forcibly terminated or other unforeseen situations occur.

Exceptions

When cases like these arise, we refer to them as **Exceptions**, which are specific and particular cases that need to be handled differently.

Exceptions

```
public void division( ) {
        Scanner scanner = new Scanner(System.in);
        double x = scanner.nextDouble();
        double y = scanner.nextDouble();
        System.out.println( "The result is " + (x / y) );
}
```

Exceptions

```
public void division( ) {
        Scanner scanner = new Scanner(System.in);
        double x = scanner.nextDouble();
        double y = scanner.nextDouble();
        System.out.println( "The result is " + (x / y) );
}           // what if x or y are not numbers?
```

Exceptions

In cases where exceptions are likely to occur, we can handle them proactively by taking precautions.

Exceptions

```
try {
        // code we want to execute
} catch ( exception_case e ) {
        // code in case of exception
}
```

Exceptions

```
try {
        // code we want to execute
} catch ( exception_case e ) {
        // code in case of exception
}
```

Exceptions

```
try {
        // code we want to execute
} catch ( exception_case e ) {
        // code in case of exception
}
```

Exceptions

```
try {
        // code we want to execute
} catch ( exception_case e) {
        // code in case of exception
}
```

Exceptions

```
try {
    double x = scanner.nextDouble();
} catch ( Exception e ){
    System.out.println("You must enter a number");
}
```

Let’s practice!