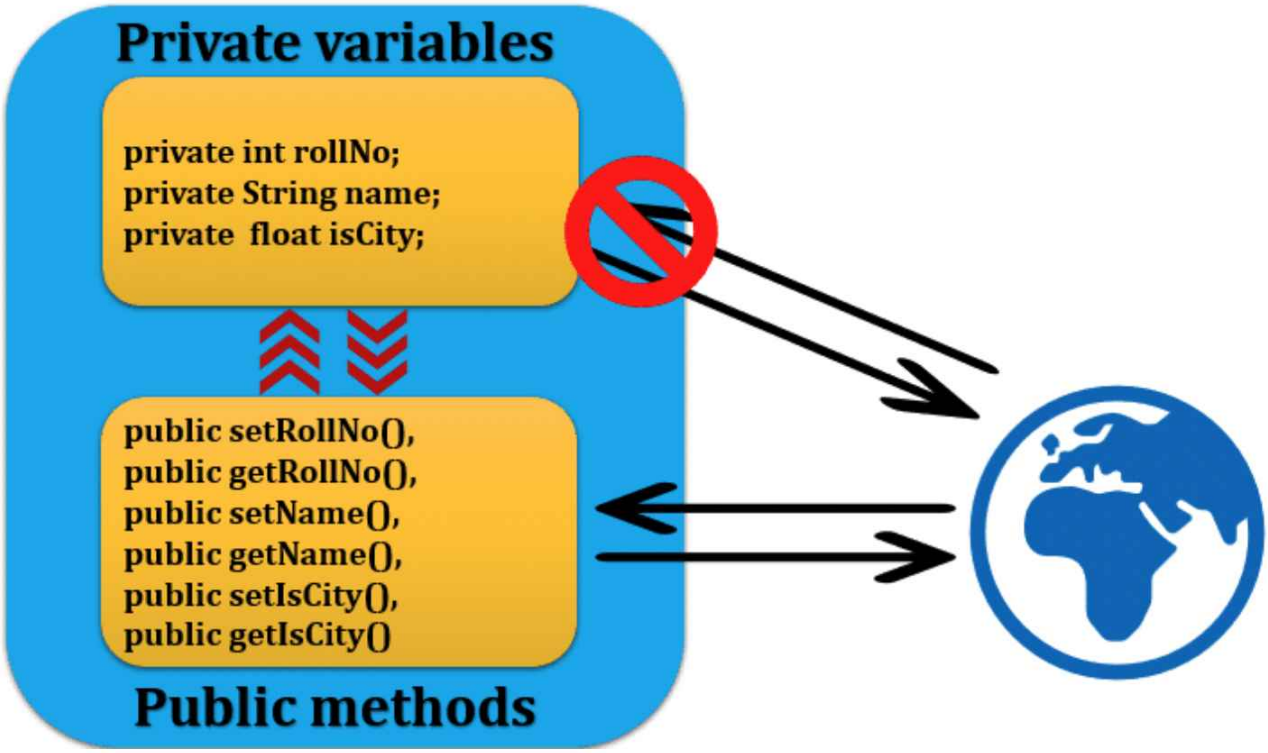
19-Encapsulation

Author: Vincent Lau

Introduction



Encapsulation is one of the four fundamental principles of object-oriented programming (OOP). It is the process of hiding the internal state and implementation details of an object and providing access to the object's data and behavior through well-defined methods. Encapsulation ensures that the object's internal state is not directly accessible from outside the class, promoting data privacy and abstraction.

In Java, encapsulation is achieved using access modifiers (e.g., `private`, `protected`, `public`) to control the visibility of class members (fields and methods).

1. Data Hiding: Encapsulation ensures that the internal state or data of an object is hidden from the outside world, protecting it from direct access or modification.
2. Access Modifiers: Encapsulation uses access modifiers (such as private, public, or protected) to control the visibility and accessibility of class members (fields and methods).
3. Encapsulating Methods: By encapsulating methods, you can provide controlled access to the underlying data, enforcing data integrity, and implementing necessary logic or validation.

Example 1: Encapsulates Class's State

- Make the instance variables (Attributes) private
- Accessing Data through Getter
- Modifying Data through Setter

```
1 public class Person {
2     private String name;
3     private int age;
```

```
4
5       public String getName() {
6           return name;
7       }
8
9       public void setName(String name) {
10          this.name = name;
11      }
12
13      public int getAge() {
14          return age;
15      }
16
17      public void setAge(int age) {
18          if (age > 0) { // validation
19              this.age = age;
20          }
21      }
22  }
```

In this example, the `Person` class **encapsulates the `name` and `age` fields** and provides access to them through getter and setter methods.

The fields are marked as private, preventing direct access from outside the class. The getter and setter methods allow controlled access to read and modify the data while enforcing data validation or logic, such as checking if the age is positive.

Example 2: Encapsulates Complex Data Structures

In this example, the `ShoppingCart` class **encapsulates a list of `Item` objects**.

- The `items` field is kept private, preventing direct modification from outside the class.
  - Make the instance type (List/ ArrayList) private.
- The class provides methods to add and remove items from the shopping cart, ensuring controlled access to the underlying list.
  - Modifying Data through Methods (List methods: add, remove).
  - Accessing Data through getter.

```
1  public class ShoppingCart {
2      private List<Item> items;
3
4      public ShoppingCart() {
5          items = new ArrayList<>();
```

```
    }

    public void addItem(Item item) {
        items.add(item);
    }

    public void removeItem(Item item) {
        items.remove(item);
    }

    public List<Item> getItems() {
        return items;
    }
}
```

Example 3: Encapsulation with Read-Only Fields

```
public class Circle {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

In this example, the `Circle` class has a private `radius` field, which is marked as `final`. The `radius` can only be set once through the constructor and cannot be modified afterward. The `getRadius()` method allows external access to the `radius` field in a read-only manner.

Example 4: Encapsulation with Private Helper Methods

```
public class StringUtil {
    public static String reverseString(String input) {
```

```
        return reverseStringHelper(input, input.length() - 1);
    }

    private static String reverseStringHelper(String input, int index) {
        if (index < 0) {
            return "";
        }
        return input.charAt(index) + reverseStringHelper(input, index - 1);
    }
}
```

In this example, the `StringUtil` class has a public static method `reverseString`, which reverses the input string. The actual recursion is implemented in the private helper method `reverseStringHelper`, which is not accessible from outside the class. This ensures that clients can use the public method without worrying about the details of the recursive implementation.

Benefits of using Objects

- **Information hiding**
  - Able to hide Inner class and not allow users to know how the data and variables are stored.
  - Able to make the fields of the class write-only (if only setter method) or read-only (if only getter method)
- **Code reuse**
  - It allows the programmer or a user to effectively use the existing code again and again.
- **Testability**
  - With encapsulation, it becomes much better for Unit testing.

Challenge: Encapsulation with Final Variable

- Is the final keyword important to make the class instance immutable?

```
public class Circle {
    private final double radius; // final is not important to make it immutable, why?

    public Circle(double radius) {
        this.radius = radius;
    }
```

```
7
8       public double getRadius() {
9           return radius;
10      }
11
12      public double calculateArea() {
13          return Math.PI * radius * radius;
14      }
15  }
```