University of Tissemsilt
Faculty of Science & Technology
Departement of Math and Computer Science

Object-Oriented Programming

Encapsulation

3 mars 2024

Lecturer

Dr. HAMDANI M

Speciality : Computer Science (ISIL)
Semester : S4

Plan

1. Understanding Classes
2. Encapsulation
3. Static variables and static methods
4. Instances in Java

1. Understanding Classes
   - Class in Java
   - Constructors in Java
2. Encapsulation
   - About Encapsulation
   - Benefits of Encapsulation
   - Encapsulation in Java
   - Modifiers
   - Accessors And Mutators
   - Access to the instance (this)
3. Static variables and static methods
   - Static Variables
   - Static Methods
   - Why Is Method main Declared static ?
4. Instances in Java
   - About Instances
   - Creating and Accessing Instances

Definition

- A class is a blueprint or template from which objects are created.
- It defines the properties (attributes/fields) and behaviors (methods/functions) that the objects created from the class will have.
- It defines a group of objects with similar characteristics (properties) and behaviors (methods).

Benefits of Using Classes

- **Encapsulation** : Classes encapsulate data and behavior, promoting modularity and code organization.
- **Reusability** : Classes can be reused in different parts of the program, reducing code duplication.
- **Abstraction** : Classes hide implementation details, allowing users to interact with objects at a higher level of abstraction.

Components of a Class

- **Fields (Attributes)** :Represent the properties or characteristics of an object (e.g., color, name, age).
- **Methods (Behaviors)** : Functions that define what an object can do. These methods can access and modify the fields of the class and perform other operations.
- **Constructors :** A special method used to initialize an object when it's created..

```java
public class Person {
  private String name;  // Fields representing
  private int age;      // Person data
  // Constructor to initialize the student
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  // Methods to access Person information
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }
  public void printInfo() {
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
  }
}
```

Constructors

- Have the same name as the class they belong to.
- Are used to initialize the state of an object.
- Do not specify a return type, not even ***void***.
- Can be overloaded (a class can have multiple constructors with different parameters).
- If no constructor is explicitly defined in a class, Java provides a default constructor automatically.
- The default constructor initializes the instance variables to their default values (e.g., ***0*** for numeric types, ***null*** for object references, ***false*** for boolean primitives)

Constructor - Example

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

  // Default constructor is created here
  public Person() {
    /* name and age will be initialized to their
    default values (null and 0, respectively) */
  }

  // Another constructor with parameters
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Some important points to note about **default constructors** :

- **Not created if other constructors exist** : If you define even one constructor explicitly, the compiler will not create a default constructor.
- **Can not be called explicitly** : You can not call the default constructor explicitly since it's not defined in your code.
- **Subclass implications** : if a subclass doesn't have any constructors explicitly defined, it will inherit the default constructor from its superclass.
  However, if the superclass doesn't have a no-argument (*default*) constructor and only has parameterized constructors, the subclass cannot have a default constructor.

Destructor

- Java does not have destructors.
- There is a mechanism called **"garbage collection"** that handles memory deallocation when objects are no longer needed
- Java Virtual Machine (JVM) automatically handles memory management and garbage collection.
- The ***finalize()*** method provides a mechanism for performing cleanup or finalization tasks before an object is garbage collected."
- ***finalize()*** is not recommended, due to its non-deterministic nature and limitations (utilize : *try-with-resources*, *close()*, *closeConnection()*, *AutoCloseable*,...)

1 Understanding Classes
- Class in Java
- Constructors in Java

2 Encapsulation
- About Encapsulation
- Benefits of Encapsulation
- Encapsulation in Java
- Modifiers
- Accessors And Mutators
- Access to the instance (this)

3 Static variables and static methods
- Static Variables
- Static Methods
- Why Is Method main Declared static ?

4 Instances in Java
- About Instances
- Creating and Accessing Instances

What is Encapsulation ?

Combines data and methods into a class.

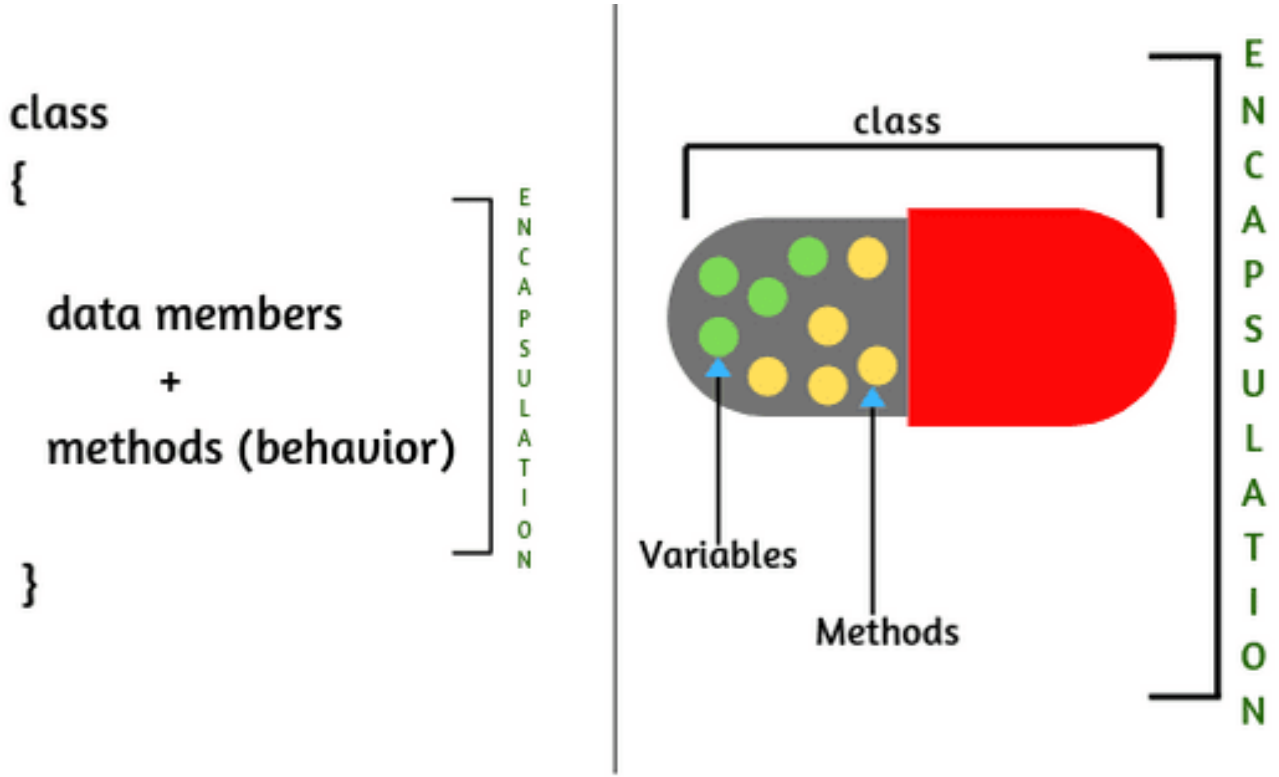

Fig: Encapsulation

Principles of Encapsulation

- Data Hiding : Encapsulation hides the internal state of objects from direct access by external code, preventing unauthorized access and manipulation.
- Access Control : Encapsulation allows you to control the visibility and accessibility of class members using access modifiers (public, private, protected, and default).
- Information Hiding : Encapsulation hides the implementation details of a class from its users, exposing only the necessary interfaces (public methods) for interacting with the class.

Benefits of Encapsulation

- Modularity : Encapsulation promotes modular design by encapsulating related data and behaviors within a single class .
- Data Integrity : Encapsulation helps maintain data integrity by providing controlled access to the internal state of objects, preventing invalid or inconsistent data states.
- Code Reusability : Encapsulation enables you to encapsulate reusable components (classes) that can be easily reused in different parts of the codebase.
- Security : Encapsulation enhances security by restricting access to sensitive data and preventing unauthorized modification of object state, protecting the integrity and confidentiality of the data.

Encapsulation in Java

- In Java, encapsulation is achieved through the use of access modifiers (public, private, protected, and default) to control the visibility and accessibility of class members.
- Getter and setter methods are used to provide controlled access to private attributes, allowing for safe and controlled manipulation of object state.

Access Modifiers

Access modifiers control method and data visibility

public : Accessible from anywhere in the program (like a master key)

private : Accessible only within the class itself (like a key for a specific room)

protected : Accessible within the class and its subclasses (like a key for a specific building)

default (no modifier) : Accessible only within the same package

Access Modifiers in Java

Access Modifier	Accessible Within Class	Accessible Within Package	Accessible in Subclass (Outside Package)	Accessible Everywhere
private	Yes	No	No	No
default	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

Java Access Modifiers and Their Accessibility

Best Practices

- Use the most restrictive access level that makes sense for a particular member. Start with ***private*** and only increase accessibility as needed.
- ***Fields*** are typically made ***private*** to enforce encapsulation.
- ***Constructors*** can be any access level, depending on whether you want to restrict instantiation of your class.
- ***Methods*** that are intended for use outside of the class should be ***public***, but internal utility methods should be ***private*** or ***default*** to limit their scope.
- ***Constant*** values (static final variables) are usually made public since they don't change and are meant to be accessible wherever needed.

Remember

By carefully choosing the appropriate access modifier for each class member, you can ensure that your Java classes expose a well-defined interface to the rest of your program, while keeping their internal implementation details hidden and protected.

Accessors (getters)

- An Accessor method is commonly known as a get method or simply a ***getter***.
- used to retrieve or access the value of an object's state (its fields or properties) from outside the class.
- Provide controlled access to internal data while maintaining encapsulation.

Getters - Accessiblity

Getter access in Java :

- **Public** (*Most Common Use*) : Widely accessible, but weakens encapsulation.
- **Protected** : Accessible within package and subclasses, good for inheritance.
- **Package-private** : Accessible within the same package, useful for internal collaboration.
- **Private** : Only accessible within the class, hides data but needs additional access methods.

Always use private fields

Getter - Example

```
public class Person {
    private String name;
    private int age;
    private boolean employed;

    public String getName() {
      return name;
    }
    public int getAge() {
      return age;
    }
    public boolean isEmployed() {
      return employed;
    }
}
```

Mutator (setters)

- Sets or updates the value (mutators) : ***setter*** is a method in java used to update or set the value of the data members or variables.
- The ***setter*** It takes a parameter and assigns it to the attribute..
- Importance : ensure data encapsulation, validate inputs, and enhance code maintainability.

Setters - Accessiblity (1)

Choose the access modifier based on context, sensitivity, and future needs.

- Public : Most common ; used when you want to allow external classes to modify the state of an object. Suitable for fields that can be safely changed by any class.
- Protected : Restricts the setting of a field to the defining class, its subclasses, and classes within the same package. Useful when you want to limit modifications to a more controlled group of classes, typically within a hierarchy.

Setters - Accessiblity (2)

- Default (Package-Private) : Allows only classes within the same package to modify the field. Good for when you're working with a set of closely related classes that need to interact more freely with each other but are not exposed to the outside.
- Private : Very rare, as setters are meant to modify the state of an object from outside. However, they can be used internally to encapsulate the setting logic within the class itself, not intended for use by any external or subclass.

Setter - Example

```
public class Person {
        private String name;
        private int age;
      public void setName(String name)
      {
        this.name = name;
      }
      public void setAge(int age) {
        if (age >= 0) {
          this.age = age;
        } else {
          throw new IllegalArgumentException("Age cannot
                be negative");
        }
      }
}
```

Naming Getters and Setters

- Use the same prefix for all getters and setters : **get**, **set**, **is**, **has**
- Use meaningful names : The name should clearly communicate the purpose of the method. Avoid generic names like *getValue*, *setValue*
- Follow the Java Naming Conventions : Use ***camelCase*** for method names
- Keep it concise : While clarity is important, avoid overly verbose names
- Consider the mutability of the property : For boolean properties, is or has prefixes can be used for getters, while set is used for setters.
- For complex properties : Consider using descriptive names : *getEmployeeInformation()*

Examples : Naming Getters and Setters

Getter names :

- getFirstName()
- getLastName()
- getSalary()
- getEmployeeInformation()
- isManager()
- isEnabled()

Setter names :

- setFirstName(String firstName)
- setLastName(String lastName)
- setSalary(double salary)
- setManager(boolean manager)

Automatically Inserting Getters and Setters

in *Netbens/Eclipse*, you can automatically generate getters and setters for your Java class fields :

- Position your cursor within the class where you want to insert the getters and setters.
- Select **Source** > **Generate Getters and Setters....**
- In the dialog that appears, Select the fields you want to generate accessors for.
- Click OK to generate the methods.
- Or, select : **Refactor** > **Encapsulate Fields...**

What is "this" ?**

- ***this*** is a reference variable that points to the current object instance.
- It is implicitly available within all instance methods and constructors.
- You can use ***this*** to access instance variables, methods, and even call other constructors within the same class.
- can be used only inside a non-static method

Uses of "this"

- Accessing instance variables : Use this to differentiate between local variables and instance variables with the same name.
- Calling other methods : Use this to call other methods on the same object, promoting modularity and code reuse.
- Calling constructors : Use this to call other constructors from within a constructor, enabling initialization with different parameters.
- Passing as an argument : You can pass this as an argument to other methods, allowing them to interact with the current object.

Example : Accessing Instance Variables

```
public class Person {
  private String name;

  public void setName(String name) {
    this.name = name;
    // this.name refers to the instance variable
  }
}
```

this.name is used within the *setName* method to differentiate the local variable name from the instance variable name

Example : Calling Other Methods

```
public class Person {
  public void greet(Person other) {
    System.out.println("Hello " + other.name + "!");
  }
  public void sayHelloTo(Person other) {
    this.greet(other); // Call greet() on the current
          object
  }
}
```

the *sayHelloTo* method uses *this.greet* to call the *greet* method on the current object instance (this). *This* allows the *sayHelloTo* method to reuse the functionality of *greet* without code duplication.

Example : Calling Constructors

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  public Person() {
    this("Dennis Ritchie");/* this() calls the
          constructor with the String argument */
  }
}
```

the no-argument constructor (*Person()*) uses *this("Dennis Ritchie")* to call the constructor that takes a name argument. *This* allows you to create objects with different initial names using different constructors.

Example : Passing "this" as an Argument

```
public class Person {
  private String name;

  public void compare(Person other) {
    if (this.age > other.age) {
      System.out.println(this.name + " is older than " +
            other.name);
    } else {
      System.out.println(other.name + " is older than " +
            this.name);
    }
  }
}
```

In this example, the *compare* method takes another *Person* object as an argument and uses *this* to access the current object's *age* and *name*.

1 Understanding Classes
- Class in Java
- Constructors in Java

2 Encapsulation
- About Encapsulation
- Benefits of Encapsulation
- Encapsulation in Java
- Modifiers
- Accessors And Mutators
- Access to the instance (this)

3 Static variables and static methods
- Static Variables
- Static Methods
- Why Is Method main Declared static ?

4 Instances in Java
- About Instances
- Creating and Accessing Instances

Static Variables

- Class variables, also known as static variables, are variables declared with the ***static*** keyword within a class.
- Belong to the class rather than instances of the class.
- Shared by all instances of the class, meaning only one copy exists (global variables).
- Accessed directly using the class name (e.g., *ClassName.variableNam*
- Can also be accessed through an object instance followed by the class name (e.g., *objectInstance.className.variableName*).
- Initialized only once at the start of the program's execution.

Accessing Class Variables

- Use the class name directly (e.g., *MyClass.count*).
- Access through an object instance (e.g., *object.MyClass.count*).
- Remember, accessing class variables through an object instance might not always reflect the latest value due to potential changes made by other objects.

Examples of Class Variables

Counter for Objects :

```
static int objectCount = 0; // to count the number of objects
```

Constants :

```
static final double PI = 3.14159;
```

Configuration Settings :

```
static String defaultLanguage = "English";
```

Database Connection Information :

```
static String databaseURL =
       "jdbc:mysql://localhost:3306/mydatabase";
 static String username = "myuser";
 static String password = "mypassword";
```

Example - Constants

```
public class Constants {
  public static final double PI = 3.14159;
  public static final int MAX_CONNECTIONS = 10;
  public static final String DEFAULT_USERNAME = "admin";
  public static final String DEFAULT_PASSWORD = "password123";
}

public class Application {
  public static void main(String[] args) {
    System.out.println("PI: " + Constants.PI);
    System.out.println("Max Connections:" +
          Constants.MAX_CONNECTIONS);
    System.out.println("Default Username: " +
          Constants.DEFAULT_USERNAME);
    System.out.println("Default Password: " +
          Constants.DEFAULT_PASSWORD);
  }
}
```

Accessing Class Variables

- Static methods are methods that belong to the class itself, not to individual objects of the class
- They are declared with the static keyword
- They can be accessed using the class name, not an object reference

```
Math.sqrt(900.0);
```

Key Characteristics of Static Methods

- Static methods cannot directly access instance variables or other non-static methods.
- They can access static variables and other static methods of the same class.
- They can be passed instance variables or other methods as arguments.
- Often used for operations that don't require any data from an instance of the class.

Static methods are like guests in a house. They can see and use the things that are publicly available in the house (static members), but they can't go into individual rooms (instance members) unless they are invited (passed as arguments)

Common Use Cases for Static Methods

- Utility methods : Perform general-purpose tasks like calculations, string manipulation, or I/O :

```
MathUtil.add(a, b); //Math operations
StringUtil.capitalizeFirstLetter(str) //String manipulation
FileUtil.readFile(fileName) //Input/Output operations
Validator.isEmailValid(email)    //Validation
```

- Constants : Define class-wide constants using : p*ublic static final*
- Factory methods : Create and return new instances of the class :

```
Integer intValue = Integer.valueOf("123");
System.out.println(intValue); // Output: 123
```

- State-independent Methods : When a method's behavior is not dependent on the state of an object

```
public static double inchesToCentimeters(double inches)
{  return inches * 2.54;   }
```

Example 01 : Static Method

```
public class TemperatureConverter {

  // Static method to convert Fahrenheit to Celsius
  public static double fahrenheitToCelsius(double fahrenheit) {
    return (fahrenheit - 32) * 5 / 9;
  }

  public static void main(String[] args) {
  /* Call the static method without creating an instance of the
        class*/
    double celsius =
          TemperatureConverter.fahrenheitToCelsius(100);
    System.out.println("100 degrees Fahrenheit is " + celsius
          + " degrees Celsius.");
  }
}
```

Example 02 : Static Method

```java
public class MathUtil {

  public static double add(double x, double y) {
    return x + y;
  }

  public static double square(double x) {
    return x * x;
  }
}

// Usage:
double result = MathUtil.add(5, 3);
double area = MathUtil.square(4);
```

Why Is Method main Declared static ? (1)

- When you execute the Java Virtual Machine (JVM) with the java command, the JVM attempts to invoke the main method of the class you specify. Declaring main as static allows the JVM to invoke main without creating an object of the class. When you execute your application, you specify its class name as an argument to the java command, as in :

  java *ClassName argument1 argument2* ...

  The JVM loads the class specified by *ClassName* and uses that class name to invoke method **main**.

Why Is Method main Declared static ? (2)

1. **Program Entry Point :** The main method serves as the entry point for your Java program. This means it's the first method that the Java Virtual Machine (JVM) executes when you run the program. Since the main method is the starting point, it's executed directly without needing an object instance of the class containing it.
2. **No Object Required** : If the main method wasn't static, you would need to create an object of the class to call the main method. However, creating an object before the program can even start wouldn't make sense. By declaring it static, the main method becomes independent of any object instances, allowing the JVM to execute it directly

Why Is Method main Declared static ? (3)

3. **Class Loading and Memory Management** : When you run a Java program, the JVM first loads the class containing the main method into memory. Since the main method is static, it's allocated in the class memory rather than in the memory of any object instance. This simplifies memory management for the JVM.

4. **Simplicity and Efficiency** : Declaring main as static keeps the code simple and efficient. You don't need to worry about creating objects or managing their lifecycle just to run the program.

5. **Consistency with Other Languages** : Many other programming languages that use object-oriented features also follow the convention of having a static main method as the program's entry point. This consistency can help programmers familiar with other languages understand how Java programs execute.

Static main execution

```
public class Greeting {

  public static void main(String[] args) {

    String firstName = args[0]; // First command-line argument
    String lastName = args[1];  // Second command-line argument

    System.out.println("Hello, " + firstName + " " + lastName
          + "!");
  }
}
```

Execution : java Greeting James Gosling

Result : Hello, James Gosling !

1 Understanding Classes
- Class in Java
- Constructors in Java

2 Encapsulation
- About Encapsulation
- Benefits of Encapsulation
- Encapsulation in Java
- Modifiers
- Accessors And Mutators
- Access to the instance (this)

3 Static variables and static methods
- Static Variables
- Static Methods
- Why Is Method main Declared static ?

4 Instances in Java
- About Instances
- Creating and Accessing Instances

What are Instances ?

- Instances are objects created from a class.
- They represent specific entities with their own state and behavior.
- Think of a class as a blueprint, and an instance as a physical object built from that blueprint
- Each instance has its own set of attributes and methods, independent of other instances of the same class.

Syntax for Creating Instances

- Instances are created using the ***new*** keyword followed by a ***constructor***.
- Constructors are special methods within a class responsible for initializing newly created objects.

```
Person prs = new Person("Kebas", 20, "Algeria");
```

Accessing Instance Variables

Dot notation (.) is used to access instance variables / methods

```
prs.greet();
prs.introduce();

System.out.println(prs.name);
  //depends on the type of access mofifier
```

Example of an instance (object)

```
class Person {
  String name;
  int age;
  String nationality;

  Person(String name, int age, String nationality) {
    this.name = name;
    this.age = age;
    this.nationality = nationality;
  }
  void greet() {
    System.out.println("Hello, my name is " + name + ".");
  }
  void introduce() {
    System.out.println("I am " + age + " years old and from "
           + nationality + ".");
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    // Create a Person instance (instantiation)
    Person prs = new Person("Kebas", 20, "Algeria");

    // Call object methods
    prs.greet();
    prs.introduce();
  }
}
```

Questions ?