Main Features of Object Oriented Programming

Encapsulation- This is the practice of keeping fields within a class private, then providing access to them via public methods. It’s a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide. Applying encapsulation uses getters and setters, where the variable name is private, but its getters and setters are public. In short, it has indirect access to values.

Example:

public class Student {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name

}

}

To access name:

Student student = new Student();

student.setName(“Dan”);

System.out.println(student.getName());

Inheritance- It lets programmers create new classes that share some of the attributes of existing classes. One common example is by using ‘extends’ keyword in a class.

class Family {

}

class Mother extends Family {

}

Polymorphism- It lets programmers use the same word to mean different things in different contexts. Examples are method overloading and method overriding. In the overloading, it can accept more than 2 types of input parameters in a single class. Like for teachers class, it can accept either 2 or 3 parameters.

public class Teacher extends Person{

public Teacher(int id, String name, int age) {

super(id, name, age);

}

public Teacher(int id, String name, int age, List<Klass> classes) {

super(id, name, age);

this.classes = classes;

}

}

For Overriding, it allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. Example is used in the exercise

@Override  
**public** String introduce() {  
 **return** String.*format*(**"%s I teach %s."**, introduce1(), (**classes** == **null** ? **"No Class"** : **"Class"** + **classes**.stream().map(klass -> **" "** + klass.getNumber()).collect(Collectors.*joining*(**","**))));  
}

Java SOLID Code Implementation

Single Responsibility- One class should only have one responsibility and it should be able to execute it well. It means that a class should be defined to its very bare requirements, like getters and setters, and do its implementation by extending it to a new class.

public class Employee {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name

}

}

public class Supervisor extends Employee {

//codes

}

Open for Extension/Closed for Modification- Modules should allow for extensions to add new features without affecting existing code to make sure the changes are confined to a smaller part of code and new bugs are not introduced to existing code. When new codes are added, there should be no major changes to the existing one.

public class Employee {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name

}

}

public class Supervisor extends Employee {

//codes

}

public class Owner extends Employee {

//NEW CODES

}

Liskov Substitution- This principle enables you to decide where to draw the line while determining behaviors of a base class. It should be defined based on functionality. For example when you add new codes, the context should align with the existing ones.

Interface Segregation- A client should never be forced to implement a function that it does not require. Instead of having bloated interfaces, segregate them based on roles.

Dependency Inversion- Entities must depend on abstractions instead of concretions. Instead of using direct references from a high-level module to a low-level module, use abstractions.