PROGRAMMING IN JAVA

Lecture 7

Encapsulation and Class Design

Access Modifiers, `this`, `static`, `final`, Immutability,
Constructor Chaining, Packages, and Modular Design

Lecturer: Arshyn Altybay

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

1. Learning Objectives

After completing this lecture, students should be able to:

1. explain the purpose of encapsulation in object-oriented programming;
2. use Java access modifiers correctly;
3. design classes with private fields and controlled public methods;
4. create getters and setters with validation;
5. explain and use the `this` keyword;
6. distinguish between instance members and static members;
7. use `final` with variables, fields, methods, and classes;
8. design simple immutable classes;
9. apply constructor overloading and constructor chaining;
10. organize classes using packages and imports;
11. use defensive copying to protect mutable internal data;
12. design small modular Java applications.

2. From Objects to Well-Designed Objects

A class may contain fields, constructors, and methods. However, simply placing data and methods inside a class does not automatically produce good object-oriented design.

Consider the following class:

```
class Student {
    public String name;
    public int age;
    public double gpa;
}
```

The program is valid Java, but every field is public. Any part of the application can assign invalid values:

```
Student s = new Student();
s.age = -500;
s.gpa = 100.0;
s.name = null;
```

The object cannot protect its own state. A better design hides internal data and controls how that data is accessed or changed. This idea is called **encapsulation**.

3. Encapsulation

3.1 Definition

Encapsulation is the object-oriented principle of combining data and behavior inside a class while restricting direct access to the internal state of an object.

A common Java design pattern is:

private fields + public methods

Private fields store the internal state. Public methods provide controlled access to that state.

3.2 Why Encapsulation Is Important

Encapsulation provides several advantages:

- **Data protection:** invalid values can be rejected.
- **Controlled access:** the class decides how its data may change.
- **Maintainability:** internal implementation can change without changing client code.
- **Modularity:** each class manages its own responsibilities.
- **Reliability:** objects can preserve valid states.
- **Reusability:** well-designed classes can be used in many programs.

4. Java Access Modifiers

Java provides several levels of access control.

Modifier	Same Class	Same Package	Subclass	Other Package
`private`	Yes	No	No	No
package-private	Yes	Yes	Limited	No
`protected`	Yes	Yes	Yes	Limited
`public`	Yes	Yes	Yes	Yes

4.1 Private Access

A private member can be accessed only inside the class in which it is declared.

```
class BankAccount {
    private double balance;
}
```

This is normally the preferred access level for fields.

4.2 Public Access

A public member can be accessed from any class that can access the containing class.

```
public void deposit(double amount) {
    // implementation
}
```

4.3 Package-Private Access

If no access modifier is written, the member has package-private access.

```
class Course {
    String title; // package-private
}
```

Such a member is accessible only from classes in the same package.

4.4 Protected Access

A protected member is accessible inside the same package and can also be available to subclasses under Java's protected-access rules. This becomes especially important when inheritance is studied.

Example 1. Encapsulating a Student Class

```java
public class Student {

    private String name;
    private int age;
    private double gpa;

    public Student(String name, int age, double gpa) {
        this.name = name;
        this.age = age;
        this.gpa = gpa;
    }

    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("GPA: " + gpa);
    }

    public static void main(String[] args) {
        Student student =
                new Student("Aruzhan", 19, 3.75);

        student.displayInfo();
    }
}
```

Expected output

```
Name: Aruzhan
Age: 19
GPA: 3.75
```

5. Getters and Setters

A **getter** returns the value of a private field. A **setter** changes the value of a private field.

```
public String getName() {
    return name;
}

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

Getters and setters are not simply a way to make private fields behave like public fields. Their main advantage is that they provide a controlled place for validation and business rules.

Example 2. Validation in Setters

```
1  public class Student {
2
3      private String name;
4      private int age;
5      private double gpa;
6
7      public Student(String name, int age, double gpa) {
8          setName(name);
9          setAge(age);
10         setGpa(gpa);
11     }
12
13     public String getName() {
14         return name;
15     }
16
17     public void setName(String name) {
18         if (name != null && !name.isBlank()) {
19             this.name = name;
20         }
21     }
22
23     public int getAge() {
24         return age;
25     }
26
27     public void setAge(int age) {
28         if (age >= 0 && age <= 150) {
```

Example 2 (continued)

```
            this.age = age;
        }
    }

    public double getGpa() {
        return gpa;
    }

    public void setGpa(double gpa) {
        if (gpa >= 0.0 && gpa <= 4.0) {
            this.gpa = gpa;
        }
    }

    public static void main(String[] args) {
        Student student =
                new Student("Dias", 20, 3.60);

        student.setAge(-10);
        student.setGpa(7.0);

        System.out.println(student.getName());
        System.out.println(student.getAge());
        System.out.println(student.getGpa());
    }
}
```

Expected output

```
Dias
20
3.6
```

The invalid age and GPA are rejected by the setter logic.

5.1 Improving Validation

In larger applications, silently ignoring invalid data may be undesirable. Later, when exception handling is studied, validation methods can throw exceptions such as `IllegalArgumentException`.

```
public void setAge(int age) {
    if (age < 0 || age > 150) {
        throw new IllegalArgumentException(
```

```
                "Invalid age: " + age);
    }
    this.age = age;
}
```

6. The `this` Keyword

The keyword `this` refers to the current object.

```
public class Student {
    private String name;

    public Student(String name) {
        this.name = name;
    }
}
```

There are two variables called `name`: the field and the constructor parameter. The expression `this.name` means the field of the current object.

The keyword `this` is commonly used to:

1. distinguish fields from parameters;
2. call another constructor in the same class;
3. pass the current object as an argument;
4. return the current object from a method.

Example 3. Returning `this`

```
public class Counter {

    private int value;

    public Counter increment() {
        value++;
        return this;
    }

    public void display() {
        System.out.println("Value = " + value);
    }

    public static void main(String[] args) {
        Counter counter = new Counter();

        counter.increment()
               .increment()
               .increment();

        counter.display();
    }
}
```

Expected output

```
Value = 3
```

This style is called **method chaining**.

7. Static Members

Normally, every object has its own copy of each instance field. A `static` field belongs to the class itself rather than to individual objects.

```
class Student {
    private static int numberOfStudents;
}
```

There is only one shared `numberOfStudents` field for the entire class.

Example 4. Static Object Counter

```
1  public class Student {
2
3      private String name;
4      private static int count = 0;
5
6      public Student(String name) {
7          this.name = name;
8          count++;
9      }
10
11     public String getName() {
12         return name;
13     }
14
15     public static int getCount() {
16         return count;
17     }
18
19     public static void main(String[] args) {
20         Student s1 = new Student("Aruzhan");
21         Student s2 = new Student("Dias");
22         Student s3 = new Student("Ali");
23
24         System.out.println(
25                 "Students created: " +
26                 Student.getCount()
27         );
28     }
29 }
```

Expected output

```
Students created: 3
```

7.1 Instance Members vs Static Members

Property	Instance Member	Static Member
Belongs to	individual object	class
Number of copies	one per object	one shared copy
Typical access	object.member	ClassName.member
Directly uses instance state	Yes	No
Typical purpose	object state	shared data / utilities

8. Static Methods

A static method belongs to the class.

```
public class MathUtility {

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

    public static void main(String[] args) {
        int result = MathUtility.square(8);
        System.out.println(result);
    }
}
```

Expected output

```
64
```

A static method cannot directly access an instance field because no particular object is implied.

9. The `final` Keyword

The keyword `final` is used in several contexts.

9.1 Final Local Variable

```
final double PI = 3.141592653589793;
```

After initialization, the variable cannot be assigned another value.

9.2 Final Instance Field

```
class Student {
    private final int id;

    public Student(int id) {
        this.id = id;
    }
}
```

Each object can have a different ID, but the ID cannot be reassigned after initialization.

9.3 Static Final Constants

Shared constants are commonly declared as:

```
public static final double VAT_RATE = 0.12;
public static final int MAX_USERS = 100;
```

By convention, constants use uppercase letters and underscores.

Example 5. Static Final Constants

```
public class Product {

    public static final double VAT_RATE = 0.12;

    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public double priceWithVat() {
        return price + price * VAT_RATE;
    }

    public static void main(String[] args) {
        Product p =
                new Product("Laptop", 500000);

        System.out.println(
                "VAT rate = " + Product.VAT_RATE
        );

        System.out.println(
                "Final price = " +
                p.priceWithVat()
        );
    }
}
```

Expected output

```
VAT rate = 0.12
Final price = 560000.0
```

10. Final References and Mutability

A final reference cannot be reassigned to another object, but the object itself may still be mutable.

```
final int[] numbers = {1, 2, 3};

numbers[0] = 100; // allowed

// numbers = new int[] {4, 5, 6}; // not allowed
```

Therefore, a final reference does not automatically make an object immutable.

11. Immutable Classes

An **immutable object** cannot change its observable state after construction.

A simple immutable class normally follows these rules:

1. fields are private;
2. fields are final when appropriate;
3. values are initialized in the constructor;
4. no setter methods are provided;
5. mutable internal objects are not exposed directly.

Example 6. Immutable Student Record

```
public final class StudentRecord {

    private final int id;
    private final String name;
    private final double gpa;

    public StudentRecord(
            int id,
            String name,
            double gpa) {

        this.id = id;
        this.name = name;
        this.gpa = gpa;
    }
```

Example 6 (continued)

```
16
17      public int getId() {
18          return id;
19      }
20
21      public String getName() {
22          return name;
23      }
24
25      public double getGpa() {
26          return gpa;
27      }
28
29      public static void main(String[] args) {
30          StudentRecord s =
31                  new StudentRecord(
32                          1001,
33                          "Aruzhan",
34                          3.85
35                  );
36
37          System.out.println(s.getId());
38          System.out.println(s.getName());
39          System.out.println(s.getGpa());
40      }
41  }
```

Expected output

```
1001
Aruzhan
3.85
```

12. Constructor Overloading and Chaining

A class may contain several constructors with different parameter lists. This is called **constructor overloading**.

A constructor may call another constructor in the same class using `this(...)`.

Important rule:

A call to `this(...)` must be the first statement in the constructor.

Example 7. Constructor Chaining

```java
public class Student {

    private String name;
    private int age;
    private double gpa;

    public Student() {
        this("Unknown", 0, 0.0);
    }

    public Student(String name) {
        this(name, 0, 0.0);
    }

    public Student(String name, int age) {
        this(name, age, 0.0);
    }

    public Student(
            String name,
            int age,
            double gpa) {

        this.name = name;
        this.age = age;
        this.gpa = gpa;
    }

    public void display() {
        System.out.println(
                name + ", " + age +
                ", GPA = " + gpa
        );
    }

    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student("Dias");
        Student s3 =
                new Student("Ali", 20, 3.5);

        s1.display();
        s2.display();
        s3.display();
```

Example 7 (continued)

```
    }
}
```

Expected output

```
Unknown, 0, GPA = 0.0
Dias, 0, GPA = 0.0
Ali, 20, GPA = 3.5
```

Constructor chaining reduces duplicated initialization code.

13. Class Invariants

A **class invariant** is a condition that should remain true for every valid object after construction and after every public operation.

For a bank account, one invariant might be:

$$\text{balance} \geq 0$$

For a student:

$$0 \leq \text{GPA} \leq 4$$

Encapsulation allows a class to enforce such rules.

Example 8. Encapsulated Bank Account

```
public class BankAccount {

    private String owner;
    private double balance;

    public BankAccount(
            String owner,
            double initialBalance) {

        this.owner = owner;

        if (initialBalance >= 0) {
            balance = initialBalance;
        }
```

Example 8 (continued)

```
    }

    public String getOwner() {
        return owner;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public boolean withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            return true;
        }
        return false;
    }

    public static void main(String[] args) {
        BankAccount account =
                new BankAccount("Dias", 100000);

        account.deposit(25000);
        boolean success = account.withdraw(50000);

        System.out.println(
                "Withdrawal: " + success
        );

        System.out.println(
                "Balance: " + account.getBalance()
        );
    }
}
```

Expected output

```
Withdrawal: true
Balance: 75000.0
```

14. Packages

A **package** is a namespace used to organize related Java classes, interfaces, and other types.

Packages help to:

- organize large projects;
- avoid class-name conflicts;
- control access;
- improve modularity;
- make source code easier to maintain.

A package declaration normally appears at the beginning of a Java source file:

```
package university.model;
```

A simple project may look like this:

Illustration

```
src/
  university/
    model/
      Student.java
      Course.java
    service/
      StudentService.java
    app/
      Main.java
```

File `Student.java`:

Example 9. Using Packages and Imports

```java
package university.model;

public class Student {

    private String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

File `Main.java`:

```java
package university.app;

import university.model.Student;

public class Main {

    public static void main(String[] args) {
        Student student =
                new Student("Aruzhan");

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

Expected output

```
Aruzhan
```

15. The `import` Statement

The `import` statement allows a class from another package to be referenced by its simple name.

Without import:

```
university.model.Student student =
        new university.model.Student("Dias");
```

With import:

```
import university.model.Student;

Student student = new Student("Dias");
```

Classes in the `java.lang` package are available automatically. Examples include `String`, `System`, `Math`, `Integer`, and `Double`.

16. Package Naming Conventions

Package names are conventionally written in lowercase letters.

```
package university.model;
package university.service;
package university.util;
package university.app;
```

In commercial software, reverse-domain naming is common:

```
package com.example.project.model;
```

17. Organizing a Small Application into Packages

A simple modular application can be organized as follows:

Illustration

```
src/
  model/
    Student.java
  service/
    StudentService.java
  util/
    GradeUtil.java
  app/
    Main.java
```

Typical responsibilities are:

- **model**: domain objects such as Student and Course;
- **service**: application logic;
- **util**: reusable helper methods;
- **app**: entry point and user interaction.

This is an early form of **separation of concerns**.

A utility class normally contains only static methods and should not be instantiated.

Example 10. Utility Class Design

```
public final class GradeUtil {

    private GradeUtil() {
        // Prevent object creation
    }

    public static String letterGrade(double score) {
        if (score >= 90) {
            return "A";
        } else if (score >= 80) {
            return "B";
        } else if (score >= 70) {
            return "C";
        } else if (score >= 60) {
            return "D";
        }

        return "F";
    }

    public static void main(String[] args) {
        System.out.println(
                GradeUtil.letterGrade(86)
        );
    }
}
```

Expected output

```
B
```

18. Defensive Copying

Encapsulation becomes more difficult when a class contains mutable objects such as arrays. Returning an internal mutable array directly allows outside code to change internal state.

A safer approach is **defensive copying**.

Example 11. Defensive Copying

```java
import java.util.Arrays;

public class StudentMarks {

    private final int[] marks;

    public StudentMarks(int[] marks) {
        this.marks = Arrays.copyOf(
                marks,
                marks.length
        );
    }

    public int[] getMarks() {
        return Arrays.copyOf(
                marks,
                marks.length
        );
    }

    public static void main(String[] args) {
        int[] original = {90, 80, 70};

        StudentMarks student =
                new StudentMarks(original);

        original[0] = 0;

        int[] copy = student.getMarks();
        copy[1] = 0;

        System.out.println(
                Arrays.toString(
                        student.getMarks()
                )
        );
    }
}
```

Expected output

```
[90, 80, 70]
```

The internal array is protected because the class copies data when receiving and returning the array.

19. Common Design Mistakes

19.1 Public Fields Everywhere

Poor design:

```
class Student {
    public String name;
    public int age;
}
```

Better design uses private fields and controlled methods.

19.2 Setters Without Validation

Poor:

```
public void setAge(int age) {
    this.age = age;
}
```

Better:

```
public void setAge(int age) {
    if (age >= 0 && age <= 150) {
        this.age = age;
    }
}
```

19.3 Making Everything Static

Static members are useful, but object-specific data should normally be instance data.

```
class Student {
    static String name;
    static double gpa;
}
```

This design would cause all Student objects to share the same name and GPA.

19.4 Exposing Mutable Internal Data

Returning internal arrays or lists directly can break encapsulation.

19.5 Very Large Classes

A class should have a clear responsibility. If one class manages students, files, databases, GUI, networking, and reports, it probably has too many responsibilities.

Example 12. A Better Student Class

```
public class Student {

    private static int nextId = 1;

    private final int id;
    private String name;
    private int age;
    private double gpa;

    public Student(
            String name,
            int age,
            double gpa) {

        id = nextId++;
        setName(name);
        setAge(age);
        setGpa(gpa);
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException(
                    "Name cannot be empty");
        }
        this.name = name;
    }
```

Example 12 (continued)

```
36
37      public int getAge() {
38          return age;
39      }
40
41      public void setAge(int age) {
42          if (age < 0 || age > 150) {
43              throw new IllegalArgumentException(
44                      "Invalid age");
45          }
46          this.age = age;
47      }
48
49      public double getGpa() {
50          return gpa;
51      }
52
53      public void setGpa(double gpa) {
54          if (gpa < 0.0 || gpa > 4.0) {
55              throw new IllegalArgumentException(
56                      "GPA must be 0..4");
57          }
58          this.gpa = gpa;
59      }
60
61      public void display() {
62          System.out.printf(
63                  "ID=%d, Name=%s, Age=%d, GPA=%.2f%n",
64                  id, name, age, gpa
65          );
66      }
67
68      public static void main(String[] args) {
69          Student s1 =
70                  new Student("Aruzhan", 19, 3.80);
71
72          Student s2 =
73                  new Student("Dias", 20, 3.45);
74
75          s1.display();
76          s2.display();
77      }
78  }
```

Expected output

```
ID=1, Name=Aruzhan, Age=19, GPA=3.80
ID=2, Name=Dias, Age=20, GPA=3.45
```

Example 13. Encapsulated Product Inventory Item

```
1  public class Product {
2
3      private static int nextId = 1000;
4      public static final int MAX_STOCK = 10000;
5
6      private final int id;
7      private String name;
8      private double price;
9      private int stock;
10
11     public Product(
12             String name,
13             double price,
14             int stock) {
15
16         id = nextId++;
17         setName(name);
18         setPrice(price);
19         setStock(stock);
20     }
21
22     public int getId() {
23         return id;
24     }
25
26     public String getName() {
27         return name;
28     }
29
30     public void setName(String name) {
31         if (name == null || name.isBlank()) {
32             throw new IllegalArgumentException(
33                     "Invalid product name");
34         }
35         this.name = name;
36     }
37
38     public double getPrice() {
39         return price;
```

Example 13 (continued)

```
    }

    public void setPrice(double price) {
        if (price < 0) {
            throw new IllegalArgumentException(
                    "Price cannot be negative");
        }
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    private void setStock(int stock) {
        if (stock < 0 || stock > MAX_STOCK) {
            throw new IllegalArgumentException(
                    "Invalid stock amount");
        }
        this.stock = stock;
    }

    public void addStock(int amount) {
        setStock(stock + amount);
    }

    public boolean sell(int amount) {
        if (amount <= 0 || amount > stock) {
            return false;
        }
        stock -= amount;
        return true;
    }

    public void display() {
        System.out.printf(
                "ID=%d, %s, Price=%.2f, Stock=%d%n",
                id, name, price, stock
        );
    }

    public static void main(String[] args) {
        Product product =
                new Product("Keyboard", 15000, 10);

```

Example 13 (continued)

```
85          product.addStock(5);
86          product.sell(3);
87          product.display();
88      }
89  }
```

Expected output

```
ID=1000, Keyboard, Price=15000.00, Stock=12
```

20. Encapsulation and Maintainability

Suppose client code uses:

```
account.deposit(10000);
```

The internal representation of `BankAccount` could later change. For example, the developer might store money using integer cents rather than a `double`. If the public method interface remains compatible, client code may not need to change.

This demonstrates an important design principle:

Depend on the public behavior of an object, not on its internal representation.

21. Guidelines for Good Class Design

When designing a Java class:

1. give the class one clear responsibility;
2. keep fields private unless there is a strong reason not to;
3. validate data at class boundaries;
4. expose behavior through meaningful methods;
5. do not automatically create setters for every field;
6. use `final` when a value should not change;
7. use `static` only for genuinely class-level data or behavior;
8. avoid exposing mutable internal objects;
9. use constructor chaining to reduce repeated code;

10. organize related classes into packages;
11. choose meaningful class, field, and method names;
12. preserve valid object states.

22. Summary

In this lecture, we learned that:

- encapsulation protects object state;
- private fields are normally accessed through controlled methods;
- getters retrieve state and setters can validate changes;
- access modifiers control visibility;
- `this` refers to the current object;
- static members belong to the class rather than individual objects;
- final variables cannot be reassigned after initialization;
- immutable objects do not change state after construction;
- constructor chaining reduces duplication;
- packages organize related Java types;
- import statements simplify access to classes from other packages;
- defensive copying protects mutable internal data;
- good class design improves maintainability, reliability, and reusability.

23. Review Questions

1. What is encapsulation?
2. Why are fields commonly declared private?
3. What is the difference between a getter and a setter?
4. Why should a setter validate data?
5. Name the four Java access levels.
6. What does package-private access mean?
7. What does the `this` keyword refer to?
8. Why is `this.name = name;` commonly used in constructors?
9. What is an instance field?
10. What is a static field?
11. How many copies of a static field exist for a class?
12. Why can a static method not directly use an instance field?
13. What does `final` mean for a local variable?
14. What does `final` mean for an instance field?
15. What is a static final constant?
16. Does a final reference make an object immutable?
17. What is an immutable object?
18. What is constructor overloading?
19. What is constructor chaining?
20. What rule applies to a call to `this(...)` inside a constructor?
21. What is a class invariant?
22. What is a package?
23. Why are packages useful?
24. What does an import statement do?
25. Which package is imported automatically by Java?
26. What is defensive copying?
27. Why is returning an internal mutable array dangerous?
28. Why is making every member static usually poor object-oriented design?
29. What is separation of concerns?

24. Programming Exercises

Beginner Level

1. Create a `Student` class with private fields `name` and `age`. Add getters and setters.
2. Add validation so age must be between 0 and 150.
3. Create a `Rectangle` class with private `width` and `height` fields.
4. Add methods `area()` and `perimeter()` to the rectangle class.
5. Create a `BankAccount` class with a private balance field.
6. Add `deposit()` and `withdraw()` methods.
7. Create a static counter that counts how many objects of a class have been created.
8. Create a static method that returns the square of an integer.
9. Create a static final constant representing a maximum value.
10. Create an immutable `Point` class with final `x` and `y` fields.

Intermediate Level

1. Create a `Product` class that rejects negative prices.
2. Create an `Employee` class in which salary can only be increased through a method.
3. Create overloaded constructors for a `Book` class and connect them using `this(...)`.
4. Create a utility class called `NumberUtil` containing at least five static methods.
5. Organize `Student`, `Course`, and `Main` into separate packages.
6. Create a `Temperature` class whose value cannot be lower than absolute zero.
7. Create an immutable `CourseInfo` class.
8. Create a class containing a private array and protect it using defensive copying.
9. Create a `LibraryMember` class that limits the number of borrowed books.
10. Create a `Counter` class whose objects have unique automatically generated IDs.

Advanced Level

1. Design an encapsulated inventory system using `Product` objects.
2. Design an immutable student transcript containing a defensive copy of an array of grades.
3. Refactor a program containing public fields into a properly encapsulated design.
4. Organize a small application into `model`, `service`, `util`, and `app` packages.
5. Create a banking model that guarantees the invariant balance ≥ 0.
6. Create a `PasswordAccount` class that never returns the stored password.
7. Build a unique registration-number generator using static members.
8. Create a configuration class containing application constants.
9. Design a `ShoppingCart` class that does not expose its internal mutable data directly.
10. Develop a small console-based university application using private fields, validation, static constants, constructor chaining, packages, and utility classes.

25. Homework

Task: Design a Modular Student Management Model

Create a small Java project with the following package structure:

Illustration

```
university/
  model/
    Student.java
    Course.java
  service/
    StudentService.java
  util/
    ValidationUtil.java
  app/
    Main.java
```

Requirements:

1. `Student` must contain an automatically generated final ID, private name, age, and GPA fields, validation methods, appropriate getters and setters, and at least two overloaded constructors.
2. `Course` must contain a final course code, course title, number of credits, and validation.
3. `StudentService` must contain methods to display a student, determine whether a student has a high GPA, and compare two students by GPA.
4. `ValidationUtil` must contain reusable static validation methods.

5. `Main` must create at least three students, create at least two courses, demonstrate getters and setters, demonstrate static members, and display program results.

Submission requirements:

- all Java source files;
- screenshots or copied console output;
- a short explanation of how encapsulation is used;
- a short explanation of which members are static and why;
- a short explanation of how packages improve the project structure.

26. Key Terms

Encapsulation
Hiding internal object state and controlling access through methods.

Access modifier
A keyword that controls visibility of a class member.

Getter
A method that returns a field value.

Setter
A method that modifies a field value, often with validation.

`this`
A reference to the current object.

Static member
A field or method that belongs to the class itself.

Final variable
A variable that cannot be reassigned after initialization.

Immutable object
An object whose state does not change after construction.

Constructor chaining
Calling one constructor from another constructor.

Package
A namespace for organizing related Java types.

Import
A declaration that allows another type to be referenced by its simple name.

Defensive copy
A copy used to prevent external code from modifying internal mutable data.

Class invariant
A condition that must remain true for every valid object.

Separation of concerns

Dividing software into components with distinct responsibilities.