UNIVERSITY OF MISSOURI-COLUMBIA
COLLEGE OF ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

3330 – OBJECT-ORIENTED PROGRAMMING

Laboratory 7: Inheritance and Polymorphism

Due Date: Friday October 17th, 2025, at 5:00 pm

1 Goal

Inheritance allows a new class to be based on an existing class. The new class inherits the members of the class it is based on. In this lab, you will learn about classes, inheritance, and polymorphisms and to practice using them in Java programs.

2 Resources

- Lecture notes "Unit 5".(it need most of unit 5 even the abstract class as well, this is note for future)

3 Directed Lab Work

3.1 Overview

3.1.1 Inheritance

Inheritance is a mechanism to derive a new class from another class (base class). The new class acquires some fields and methods of the base class. Inheritance is one of the important principles of the object-oriented programming, allowing developers to build convenient class hierarchies and reuse the existing code.

There are several terms. A class that is **derived** from another class is called a **subclass** (also it's known as a **derived class**, **extended class** or **child class**). The class from which the **subclass** is derived is called **superclass** (also a **base class** or a **parent class**).

To derive a new class from another class the keyword **`extends`** is used. The common syntax is shown below:

```
class SuperClass { }
class SubClassA extends SuperClass { }
class SubClassB extends SuperClass { }
class SubClassC extends SubClassA { }
```

Some important points:

- Java doesn't support multiple-classes inheritance that means a class can inherit from a single superclass;
- a class hierarchy can have multiple levels (class **C** can extend class **B** that extend class **A**);
- a superclass can have more than one subclasses.

A subclass inherits all non-private fields and methods from the superclass. Also, a subclass can add new fields and methods. The inherited and added members will be used in the same way.

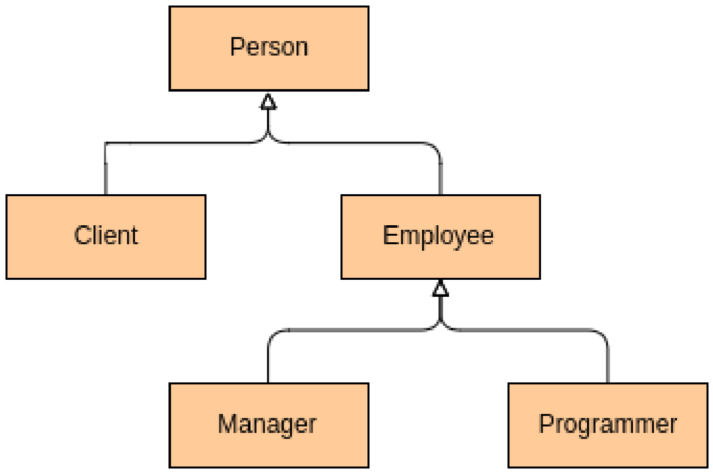
A subclass doesn't inherit private fields and methods from the superclass. However, if the superclass has public or protected methods for accessing its private fields, these members can be used inside the subclasses. Constructors are not inherited by subclasses, but the superclass's constructor can be invoked from the subclass using the special keyword `super`.

If you'd like the base class members to be accessible from all subclasses but not accessible from outside code (excludes code in the same package), use the access modifier `protected`. **Inheritance** represents the **IS-A** relationship. A base class represents a general thing and subclass represents a particular thing.

3.1.2 An Example of Class Hierarchy

Let's consider a more useful sample. A telecommunication company serves clients. It has a small staff consisting only of managers and programmers. Here is a class hierarchy for people connected with the company's activities including clients to store them in a database. This diagram shows the hierarchy for these people. An arrow means one class extends another one.

- the base class Person has fields for storing common data: name, year of birth, and address;
- the class Client has additional fields to store contract number and status (gold or not);
- the class Employee stores the start date of work for the company and salary;
- the class Programmer has an array of the used programming languages;
- the class Manager may have a dazzling smile.



Let's see the code: (next page)

```
class Person {
    private String name;
    private int yearOfBirth;
    private String address;
    // public getters and setters for all fields here
}
class Client extends Person {
    private String contractNumber;
    private boolean gold;
    // public getters and setters for all fields here
}
class Employee extends Person {
    private Date startDate;
    private Long salary;
    // public getters and setters for all fields here
}
class Programmer extends Employee {
    private String[] programmingLanguages;
    public String[] getProgrammingLanguages() {
        return programmingLanguages;
    }
    public void setProgrammingLanguages(String[] programmingLanguages) {
        this.programmingLanguages = programmingLanguages;
    }
}
class Manager extends Employee {
    private boolean smile;
    public boolean isSmile() {
        return smile;
    }
    public void setSmile(boolean smile) {
        this.smile = smile;
    }
}
```

This hierarchy has two levels and five classes overall. All fields are protected that means they are visible to subclasses. Also, each class has public getters and setters but some are skipped for short.

Let's create an object of the Programmer class and fill the inherited fields using the inherited setters. To read values of the fields we can use inherited getters.

```
Programmer p = new Programmer();
p.setName("John Elephant");
p.setYearOfBirth(1985);
p.setAddress("Some street, 15");
p.setStartDate(new Date());
p.setSalary(500_000L);
p.setProgrammingLanguages(new String[] { "Java", "C++", "Python" });
System.out.println(p.getName()); // John Elephant
System.out.println(p.getSalary()); // 700000
System.out.println(Arrays.toString(p.getProgrammingLanguages())); // [Java, C++, Python]
```

We also can create an instance of any class included in the considered hierarchy.

So, inheritance provides a powerful mechanism for code reuse and writing convenient hierarchies. Many things of the real world can be simulated like hierarchies from general to particular concept.

3.1.3 Final classes

If the class declared with the keyword **`final`**, it cannot have subclasses.

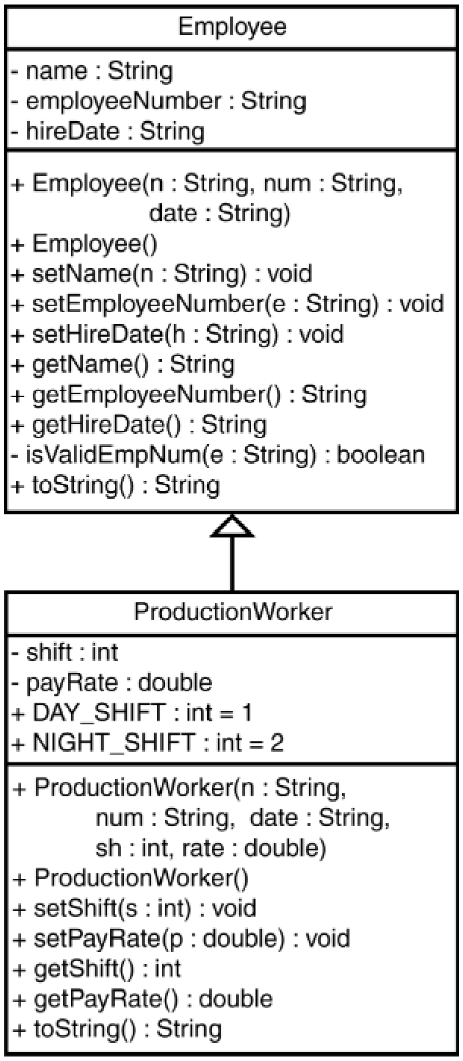
```
final class SuperClass { }
```

If you'll try to extend the class, the compile-time error will happen.

Some standard classes are declared as final: **`Integer`**, **`Long`**, **`String`**, **`Math`**. They cannot be extended:

3.2 Task 1

- Use IntelliJ IDE to create a new Java project named Task1.
- Create two new Java classes named **Employee** and **ProductionWorker**.
- These classes should be implemented according to the information represented by the following UML diagram.
- The Employee class should have fields to hold the following information:
- Employee name
- Employee number
- Hire date
- The class ProductionWorker inherits from the Employee class and has fields to hold the following information:
- Shift (an integer)
- Hourly pay rate (a double)
- The workday is divided into two shifts: day and night. The shift field will be an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2.
- The toString() method in both classes need to be developed so that they produce the output exactly as shown below.
- Create another class called **Task1Demo.** The main method of this class creates two objects of type ProductionWorker named workerOne and workerTwo. Then it prints these objects information using the statements: System.out.println(workerOne) and System.out.println(workerTwo)
- The workerOne and workerTwo object need to be initialized so that the following information will be displayed when we print these objects.


Employee
- name : String
- employeeNumber : String
- hireDate : String
+ Employee(n : String, num : String, date : String)
+ Employee()
+ setName(n : String) : void
+ setEmployeeNumber(e : String) : void
+ setHireDate(h : String) : void
+ getName() : String
+ getEmployeeNumber() : String
+ getHireDate() : String
- isValidEmpNum(e : String) : boolean
+ toString() : String
ProductionWorker
- shift : int
- payRate : double
+ DAY_SHIFT : int = 1
+ NIGHT_SHIFT : int = 2
+ ProductionWorker(n : String, num : String, date : String, sh : int, rate : double)
+ ProductionWorker()
+ setShift(s : int) : void
+ setPayRate(p : double) : void
+ getShift() : int
+ getPayRate() : double
+ toString() : String

The first production worker.
Name: John Smith

```
Employee Number: 123-A
Hire Date: 11-15-2005
Shift: Day
Hourly Pay Rate: $16.50

The second production worker.
Name: Joan Jones
Employee Number: 222-L
Hire Date: 12-12-2005
Shift: Night
Hourly Pay Rate: $18.50
```

- Don't forget to add comments.

3.3 Overriding instance methods

Java provides an ability to declare a method in a subclass with the same name as a method in the superclass. It's known as method overriding. The benefit of overriding is that a subclass can give its own specific implementation of a superclass method. Overriding methods in subclasses allows a class to inherit from a superclass whose behavior is "close enough" and then to change this behavior as the subclass needed. Instance methods can be overridden if they are inherited by the subclass. The overriding method must have the same name, parameters (number and type of parameters), and the return type (or a subclass of the type) as the overridden method.

Example. Here is an example of overriding.

```
class Mammal {
    public String sayHello() {
        return "ohlllalalalalalaoaoaoa";
    }
}
class Cat extends Mammal {
    @Override
    public String sayHello() {
        return "meow";
    }
}
class Human extends Mammal {
    @Override
    public String sayHello() {
        return "hello";
    }
}
```

The hierarchy includes three classes: Mammal, Cat and Human. The class Mammal has the method sayHello. Each subclass overrides the method. The **@Override** annotation indicates the method is overridden. It is optional but helpful.

Let's create instances and invoke the method.

```
Mammal mammal = new Mammal();
System.out.println(mammal.sayHello()); // it prints "ohlllalalalalalaoaoaoa"
Cat cat = new Cat();
System.out.println(cat.sayHello()); // it prints "meow"
Human human = new Human();
System.out.println(human.sayHello()); // it prints "hello"
```

As you can see, each subclass has its own implementation of the method sayHello.

Note: you can invoke the base class method in the overridden method using the keyword **super**.

3.3.1 Rules for overriding methods

There are several rules for methods of subclasses which should override methods of a superclass:

- the method must have the same name as in the superclass;
- the arguments should be exactly the same as in the superclass method;
- the return type should be the same or a subtype of the return type declared in the method of the superclass;
- the access level must be the same or more open than the overridden method's access level;
- a private method cannot be overridden because it's not inherited by subclasses;
- if the superclass and its subclass are in the same package, then package-private methods can be overridden;
- static methods cannot be overridden

To verify these rules, here is a special annotation @Override. It allows you to know whether a method will be actually overridden or no. If for some reason, the compiler decides the method cannot be overridden, it will generate an error. But, remember, the annotation is not required, it's only for convenience.

3.3.2 Forbidding overriding

If you'd like to forbid an overriding of a method, declare it with the keyword **`final`**.

```
public final void method() {
    // do something
}
```

Now, If you try to override this method in a subclass, a compile-time error happens.

3.3.3 Overriding and overloading methods together

It's possible to override and overload an instance method in a subclass at the same time. Overloaded methods do not override superclass instance methods. They are new methods, unique to the subclass.

The following example demonstrates it.

```
class SuperClass {
    public void invokeInstanceMethod() {
        System.out.println("SuperClass: invokeInstanceMethod");
    }
}
class SubClass extends SuperClass {
    @Override
    public void invokeInstanceMethod() {
        System.out.println("SubClass: invokeInstanceMethod is overridden");
    }

    // @Override -- method doesn't override anything
    public void invokeInstanceMethod(String s) {
        System.out.println("SubClass: overloaded invokeInstanceMethod");
    }
}
```

The following code creates an instance and calls both methods:

```
SubClass clazz = new SubClass();
clazz.invokeInstanceMethod();   //SubClass: invokeInstanceMethod() is overridden
clazz.invokeInstanceMethod("s");//SubClass: overloaded invokeInstanceMethod(String)
```

Remember, overriding and overloading are different mechanisms, but you can mix them together in one class hierarchy.

3.3.4 Hiding static methods

Static methods cannot be overridden. If a subclass has a static method with the same signature (name and parameters) as a static method in the superclass then the method in the subclass hides the one in the superclass. It's completely different from methods overriding.

You will get a compile-time error if a subclass has a static method with the same signature as an instance method in the superclass or vice versa. But if the methods have the same name but different parameters there are no problems.

3.4 Kinds of polymorphism

In the general case, polymorphism means that something (an object or another entity) has many forms. Java provides two types of polymorphism: **static (compile-time)** and **dynamic (run-time)** polymorphism. The first one is achieved by method **overloading**, the second one based on inheritance and method **overriding**.

In this lab, we'll consider only run-time polymorphism that is widely used in the object-oriented programming

3.4.1 Runtime polymorphic behavior

We remind you, method overriding it is when a subclass redefines a method of the superclass with the same name. The run-time polymorphism relies on two principles:

- a reference variable of a superclass can refer to any subtype objects;
- a superclass method can be overridden in a subclass.

It works when an overridden method is calling through the reference variable of a superclass. Java determines at runtime which version (superclass/subclasses) of the method is to be executed based on the type of the object being referred, not the type of reference. It uses the mechanism known as **dynamic method dispatching**.

Example. Here, a class hierarchy is presented. The superclass `MythicalAnimal` has two subclasses: `Chimera` and `Dragon`. The base class has the method `hello`. Both subclasses override the method.

```
class MythicalAnimal {
    public void hello() {
        System.out.println("Hello, I'm an unknown animal");
    }
}
class Chimera extends MythicalAnimal {
    public void hello() {
        System.out.println("Hello! Hello!");
    }
}
class Dragon extends MythicalAnimal {
    public void hello() {
        System.out.println("Rrrr...");
    }
}
```

We can create references to the class MythicalAnimal and assign it to subclass objects.

```
MythicalAnimal chimera = new Chimera();
MythicalAnimal dragon = new Dragon();
MythicalAnimal animal = new MythicalAnimal();
```

And invoke overridden methods through the base class references:

```
animal.hello(); // Hello, i'm an unknown animal
chimera.hello(); // Hello! Hello!
dragon.hello(); // Rrrr...
```

So, the result of a method call depends on the actual type of an instance, not a reference. It's a polymorphic feature in Java. The JVM calls the appropriate method for the object that is referred to in each variable.

It allows a class to specify methods that will be common to all of its subclasses while allowing subclasses to define the specific implementation of some or all of those methods. It's very useful for object-oriented design, especially, together with abstract methods and interfaces.

3.4.2 Polymorphism within a class hierarchy

The same thing works with methods that are used only within a hierarchy and not accessible from outside.

Example. There is a hierarchy of files. The parent class `File` represents a description of a single file in the file system. It has a subclass named `ImageFile`. It overrides the method `getFileInfo` of the parent class.

```
class File {
    protected String fullName;
    // constructor with a single parameter
    // getters and setters
    public void printFileInfo() {
        String info = this.getFileInfo(); // here is polymorphic behaviour!!!
        System.out.println(info);
    }
    protected String getFileInfo() {
        return "File: " + fullName;
    }
}
class ImageFile extends File {
    protected int width;
    protected int height;
    protected byte[] content;
    // constructor
    // getters and setters
    @Override
    protected String getFileInfo() {
        return String.format("Image: %s, width: %d, height: %d", fullName, height,
width);
    }
}
```

The parent class has the public method `printFileInfo` and the protected method `getFileInfo`. The second method is overridden in the subclass, but the subclass doesn't override the first method.

Let's create an instance of ImageFile and assign it to the variable of File.

```
File img = new ImageFile("img.png", 640, 480, someBytes); // assigning an object
```

Now, when we call the method `printFileInfo`, it invokes the overridden version of the method `getFileInfo`.

```
img.printFileInfo(); // It prints "Image: img.png, width: 480, height: 640"
```

So, run-time polymorphism allows you to invoke an instance overridden method of a subclass having a reference to a base class.

3.5 Task 2

- Use IntelliJ IDE to create a new Java project named Task2.
- Create a new Java class called **Animal.** This class has a method say() which returns an empty string.
- In the same project, create three new classes Cat, Dog, and Duck. These classes inherit from the class Animal.
- The class Cat overrides the method say() so that it returns the string "meow-meow"
- The class Dog overrides the method say() so that it returns the string "arf-arf"
- The class Duck overrides the method say() so that it returns the string "quack-quack"
- In the same project, create another class called **Task2Demo.** The main method of this class creates three objects of type Animal named cat, dog, and duck. cat should be an instance of Cat, dog is an instance of Dog, and duck is an instance of Duck. The main method will then invoke the method say() using these three objects and prints their return values using System.out.println.
  - Sample Output:

```
Cat says:
meow-meow

Dog says:
arf-arf

Duck says:
quack-quack
```

- Run your program and make sure it prints the correct output.
- Don't forget to add comments.

3.6 Task 3

- In this task you will be writing the quiz part of this lab. Use the Tests & Quizzes tab on Canvas to complete and submit answer for this quiz.
- The quiz includes one programming question to evaluate your understanding of this lab material. You will be given **30 minutes** to complete and submit your answer, please follow the following important instructions.
  - You are to complete this quiz independently and alone. Collaborating, discussing, or sharing of information during the quiz is not permitted.
  - Use the IntelliJ Idea IDE to create a new Java project file and named it yourUMId_Lab7_Quiz. yourUMId is the first part of your UM (University of Missouri) email address before the @ sign.
  - The quiz link will be inactive on **October 11 at 5:00 PM**. Note that, it is allowed to submit only once, so please review your code carefully before submitting your answer.
  - To start, click Final Assessment link. Once you click "Begin Assessment," you will have 30 minutes to complete this quiz.
  - Read the question and start to write your codes using the opened project file in the IntelliJ IDE.
  - Once you finished writing your program, export your project into yourUMId_Lab7_Quiz.zip.
  - To submit your answers, click the Browse button, navigate the file explorer to your exported zip file, click Upload button, and then click Submit for Grading button.

4 Hand In

- Create a new folder and name it yourUMId_Lab7, then copy the folders of projects from Task1 and Task2 in your new folder yourUMId_Lab7. yourUMId is the first part of your UM (University of Missouri) email address before the @ sign Zip yourUMId_Lab7 folder into yourUMId_Lab7.zip.
- **Please note that the naming conventions, the types, and the submitted file structure are very important. The improper file structures, types or names will cause losing marks.**
- Submit your *yourUMId_* Lab7.zip file into Canvas on **October 17, before 5:00 PM**.
- **No submissions will be accepted after October 17, 5:00 PM.**