CSEN401 – Computer Programming Lab

Topics:
Object Oriented Features: Inheritance and Encapsulation

Prof. Dr. Slim Abdennadher

Object-Oriented Paradigm: Features

Easily remembered as **A-PIE**

- **A**bstraction
- **P**olymorphism
- **I**nheritance
- **E**ncapsulation

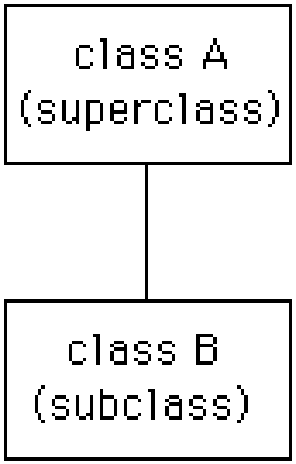
Object-Oriented Paradigm: Features

- **Inheritance**: Objects can be defined and created that are specialized types of already-existing objects.
- **Polymorphism**: the ability of objects belonging to different types to respond to method calls to methods of the same name, each one according to the right type-specific behavior.
- **Abstraction**: the ability of a program to ignore the details of an object's (sub)class and work at a more generic level when appropriate.
- **Encapsulation**: Ensures that users of an object cannot change the internal state of the object in unexpected ways.

Inheritance

Inheritance

A way to describe family of types when one **subclass** (child) "inherits" instance variables and instance methods from a **superclass** (parent).



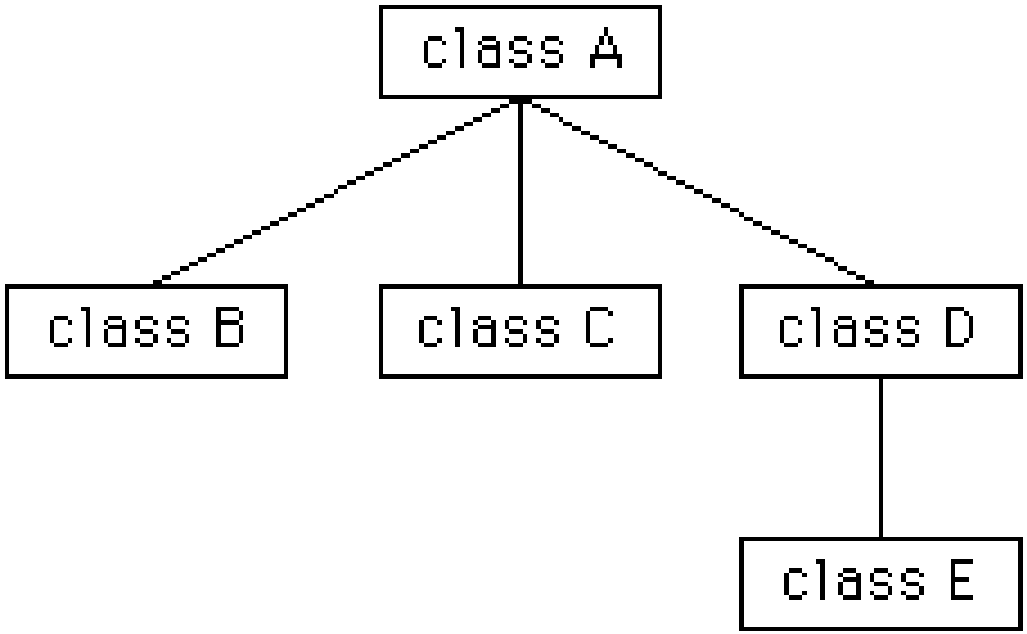
- **Base class** specifies similarities.
- **Derived class** specifies differences.
- **Advantage**: It enables the programmer to change the behavior of a class and add functionality without rewriting the entire class from scratch.
- A subclass contains more methods than superclass.
- In Java, you can only inherit from one superclass: **Single Inheritance**.

Inheritance in Java

- To **define** a class `B` to be a subclass of a class `A`:

```
class B extends A {
      .... }
```

- **Several classes** can be declared as subclasses of the same superclass.
- Inheritance can also extend over several **„generations“** of classes.



A Base Class: Person

```
public class Person
{
    private String name;

    public Person() {
       name = "No name yet";
    }
    public Person(String initialName) {
       name = initialName;
    }
    public void setName(String newName) {
       name = newName;
    }
    public String getName() {
       return name;
    }
}
```

Inheritance, Java, and Constructors I

- To call the constructor of a superclass in a subclass use the `super()` function.

```
public class Student extends Person
{
    private int studentNumber;
    public Student() {
       super();
       studentNumber = 0;
    }
    public Student(String initialName, int initialStudentNumber) {
       super(initialName);
       studentNumber = initialStudentNumber;
    }
}
```

- `super` must always be the **first** action taken in a constructor definition.
- It cannot be used later in the definition.

Inheritance, Java, and Constructors

When defining a constructor for a class, you can use `this` as a name for another constructor in the same class.

```
public Student(String initialName) {
   this(initialName, 0);
}
```

Overriding Method Definitions

- The definition in a derived class is said to **override** the definition in the base class if
  - the derived class defines a method with the **same name** as a method in the base class and
  - the method has the **same number and types** of parameters as a method in the base case.
- When overriding a method, you can change the body of the method definition to anything you wish.
- The return type of the overriding method definition must be the **same** as the return type for the method in the base class.

Call to an Overridden Method

- Assume you have the following method in the `Person class`:

```
public void writeOutput() {
    System.out.println("Name: " + name);
}
```

- If you would like to write a method `writeOutput` for the class `Student`, you can call the method `writeOutput` of the class `Person` by **prefacing** the method name with `super` and a dot:

```
public void writeOutput() {
    super.writeOutput();
    System.out.println("Student Number " + studentNumber);
}
```

- What is the difference between **Overloading** and **Overriding**?

Encapsulation

Encapsulation makes an object look like a black box.

- The insides of the box are hidden from view.
- Controls are on the outside of the box.
- The user can change the operation of the box only by using the controls.

Encapsulation

- Object oriented Programming languages make use of **encapsulation** to enforce the **integrity of a type** (i.e. to make sure data is used in an appropriate manner) by preventing programmers from accessing data in a non-intended manner.
- Through encapsulation, only a predetermined group of functions can access the data.
- The collective term for datatypes and operations (methods) bundled together with access restrictions (`public`/`private`, etc.) is a **class**.

The `private` Access Modifier and Access Methods

- When a member of a class is declared `private` it can be used only by the methods of **that class**.
- **Access methods** are methods which uses the private data of their object and are visible to other classes.
- When data is `private` the only changes to it are made through a small number of access methods.
- This helps keep objects **consistent** and **bug-free**. If a bug is detected, there are only a few places to look for it.

The private Access Modifier and Access Methods – Example

```
class CheckingAccount
{
  private String accountNumber;
  private String accountHolder;
  private int    balance;

  int currentBalance()
  {
    return balance ;
  }

  void  processDeposit( int amount )
  {
    balance = balance + amount ;
  }
  ....
}
```

Private Methods

- A `private` **method** of an object can be used only by the other methods of the object.
- Parts of a program outside of the object cannot directly use a private method of the object.
- **Example:**

```
class CheckingAccount
{ private int    balance;
  private int    useCount = 0;

  private void incrementUse()
  { useCount = useCount + 1; }

  void  processDeposit( int amount )
  { incrementUse();
    balance = balance + amount ;
  }
}
```

The `public` Access Modifier

- The **`public` access modifier** explicitly says that a method or variable of an object can be accessed by code outside of the object, i.e. in any other class or subclass.
- The `public` access modifier is usually used for all **access methods** and **constructors** in a class definition.
- **Hint**: Most variables are made `private`.

The `protected` Access Modifier

- A **`protected` member** can be accessed from the same class, a sub-class, and the same package.
- A **package** is a group of related classes.
- In the labs, we will discuss how to create a package of classes.
- A `protected` member cannot be accessed from different packages.

Default Visibility

- If you do not specify `public`, `private` or `protected` for a variable or a method, then it will have **default visibility**.
- Default visibility allows a variable or method to be seen by all methods of a class or other classes that are part of the same package.

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.
- The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

Summary of Different Access Modifiers

Keyword	Effect
`private`	Members are not accessible outside the class. A private constructor: The class cannot be instantiated. A `private` method can only be called from within the class.
`public`	Members are accessible anywhere the class is accessible. A class can be given either package or public access.
None	Members are accessible from classes in the same package only. A class can be given either package access or public access
`protected`	Members are accessible in the package and in subclasses of this class. Note that `protected` is less protected than the default package access.

Inheritance and Encapsulation: Questions

- Are private class-level variables inherited?
- How could you access the private class-level variables in subclasses?