Lecture 4

Class Design & Encapsulation In Object-Oriented Programming

Review OOP (Part I)

- In the previous lecture, you learned about:
  - Why use OOP?
  - Object, attributes and behavior
  - Class
  - Instances in Java
  - Class as a 3-compartment box
  - The dot (`.`) operator

Member Variables

- A member variable has a name (or identifier) and a type; and holds a value of that particular
  - **Naming Convention:** A variable name shall be a *noun* or *noun phrase*. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case).

```
[AccessControlModifier] type variableName [= initialValue];
[AccessControlModifier] type variableName-1 [= initialValue-1]
[, type variableName-2 [= initialValue-2]] ... ;
```

E.g:

```
private double radius;
public int length = 1, width = 1;
```

Member Methods

- A method:
  - receives arguments from the caller,
  - performs the operations defined in the method body, and
  - returns a piece of result (or void) to the caller.
- The syntax for method declaration in Java is as follows:

```
[AccessControlModifier] returnType methodName ([parameterList]) {
   // method body or implementation
   ......
}
```

Member Methods

- Method Naming Convention:
  - A method name shall be a verb, or a verb phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case).
  - For example, `getArea()`, `setRadius()`…

```
// return the area of this Circle instance
public double getArea() {
   return radius * radius * Math.PI;
}
// return type is void => does not contain a return value
public void setWidth(double width) {
      this.width = width;
}
```

Method Overloading

- Method overloading allows a method name to have multiple implementations (version).
- These implementations must be distinguishable by their parameter list.
- Parameters can differ in number, type, or order.

Method Overloading

```
public class MyClass{
    void method(){...}
    void method(int x){...}
    void method(float x){...}
    void method(int x, double y){...}
}
```

- In class `MyClass`, there are 4 methods with the same name but different parameters

Method Overloading

- **Example**: The method `average()` has 3 versions, with different parameter lists. The caller can invoke the chosen version by supplying the matching arguments.

```
public class MethodOverloadingTest {
   public static int average(int n1, int n2) { // version 1
      System.out.println("Run version 1");
      return (n1 + n2) / 2;
   }
   public static double average(double n1, double n2) {
      // version 2
      System.out.println("Run version 2");
      return (n1 + n2) / 2;
   }
   public static int average(int n1, int n2, int n3) { // version 3
      System.out.println("Run version 3");
      return (n1 + n2 + n3) / 3;
   }
}
```

Method Overloading

```
public static void main(String[] args) {
    System.out.println(average(1, 2));
    // run version 1
    // 1
    System.out.println(average(1.0, 2.0));
    // run version 2
    // 1.5
    System.out.println(average(1, 2, 3));
    // run version 3
    // 2
    System.out.println(average(1.0, 2));
    // run version 2 (int 2 implicitly casted to double 2.0)
    // 1.5
    // average(1, 2, 3, 4);
    // compilation error: no suitable method found
    // for average(int,int,int,int)
}
```

Constructors

- A constructor is a special method used to create objects
- Characteristics of a constructor:
  - It has the same name as the class
  - It has no return type in its method heading. It implicitly returns `void`
  - It does not return a value
  - It can only be invoked via the `new` operator
  - Constructors are not inherited (explained later)

Example of constructor

```
public class Circle {
    private double radius;
    private String color;
    public Circle() {
        radius = 1.0;
        color = "red";
    }
    public Circle(double r) {
        radius = r;
        color = "red";
    }
    public Circle(double r, String c) {
        radius = r;
        color = c;
    }
}
```

Class

```
Circle c1 = new Circle();
// use 1st constructor
Circle c2 = new Circle(2.0);
// use 2nd constructor
Circle c3 = new Circle(3.0, "red");
// use 3rd constructor
```

Instances

Contructor Overloading

- Constructor, like an ordinary method, can also be overloaded.
  - The above `Circle` class has three overloaded versions of constructors differentiated by their parameter list, as followed:

```
Circle() // the default constructor
Circle(double r)
Circle(double r, String c)
```

Overloading

- Depending on the actual argument list used when invoking the method, the matching constructor will be invoked.
- If your argument list does not match any one of the methods, you will get a compilation error.
  - **Note:** C language does not support method overloading. You need to use different method names for each of the variations. C++, Java, C# support method overloading.

`this` keyword

- `this` is used to refer to this instance inside a class definition.
- `this` is used within a class to reference the members of the class **(fields and methods)**
- Use `this.field` to differentiate between the class's field and local variables or method parameters

`this` keyword

```
public class Circle {
   double radius;
   public Circle(double radius)
      this.radius = radius;
      // "radius = radius" does not make sense!
      // "this.radius" refers to an instance's member
      // "radius" resolved to the method's parameter
   }
   ...
}
```

- Java offers a keyword called `this` to address such naming conflicts.
- Using `this.radius` specifies the member variable, whereas simply `radius` refers to the method's argument.

Package

- Package is used to divide classes and interfaces into different groups.
- This job is similar to putting files in folders on the hard disk
- The following example creates class `MyClass` in package `lect4`;

```
package lect4;

public class MyClass {

}
```

Package

- Similar to hard disk files & folders structure, we can create sub-packages inside packages
- In Java there are many packages divided by function
  - `java.util`: contains utility classes
  - `java.io`: contains data input/output classes
  - `java.lang`: contains commonly used classes…

Importing package

- The import statement is used to indicate that the class has been defined in a package
- Classes in the `java.lang` package and classes defined in the same package as the using class will be imported by default

```
package lect.lect4;
import lect.lect3.MyClass;
import java.util.Scanner;
public class HelloWorld {
        public static void main(String[] args) {
        MyClass obj = new MyClass();
        Scanner sc = new Scanner(System.in);
… }}
```

The Access Control Modifiers

- An access control modifier can be used to control the visibility of a class, or a member variable or a member method within a class
- In java there are 4 different characteristics:
  - `private`: allowed to be used only internally within the class
  - `public`: completely public
  - {default}:
    - It is public for classes accessing the same package
    - Is private from other export packages of the classes
  - `protected`: similar to {default} but allowed to design even if the subclass and package are different

The Access Control Modifiers

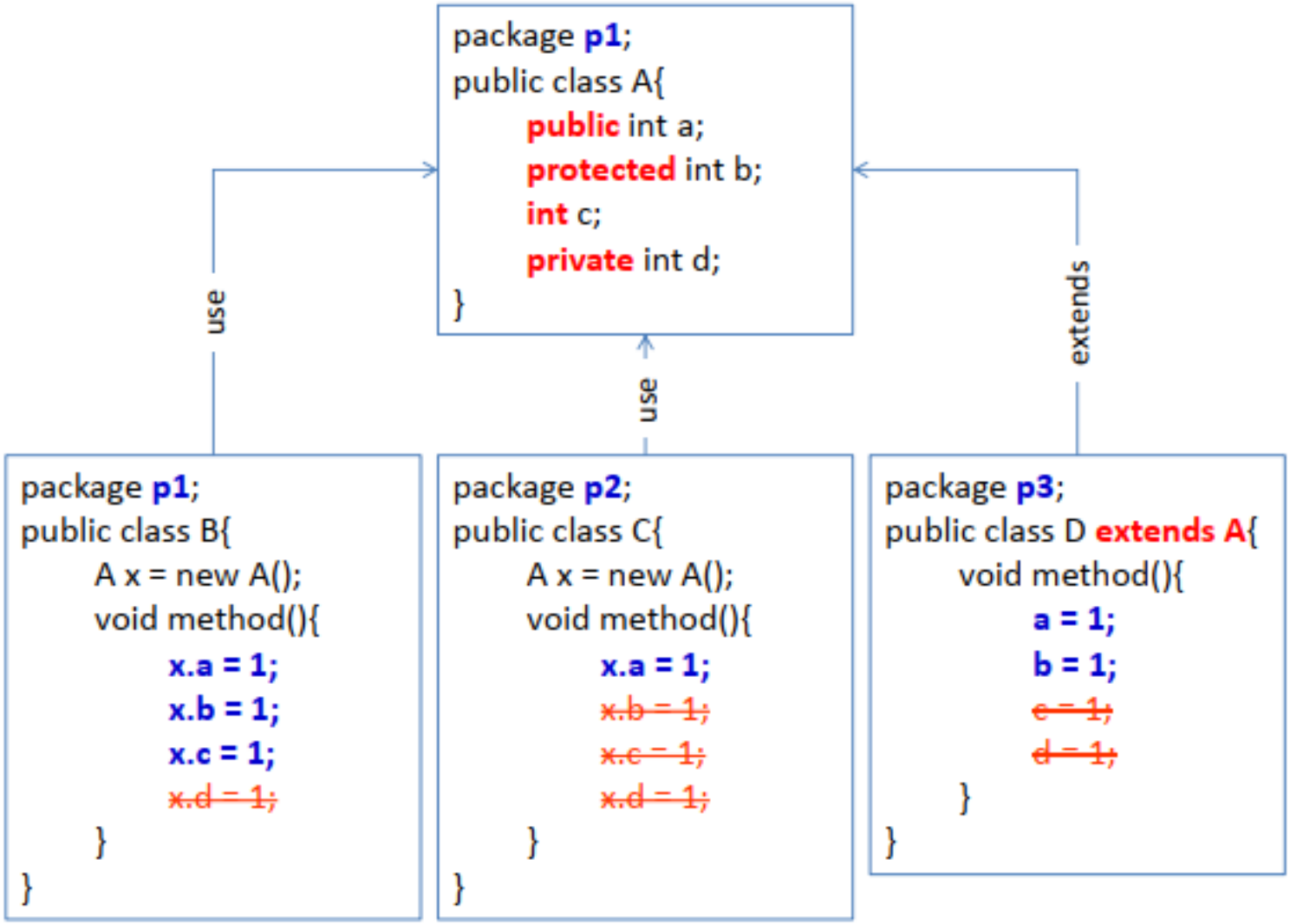
- The level increases gradually in the direction of the arrow

public → protected → {default} → private

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

The Access Control Modifiers



Non-Encapsulation

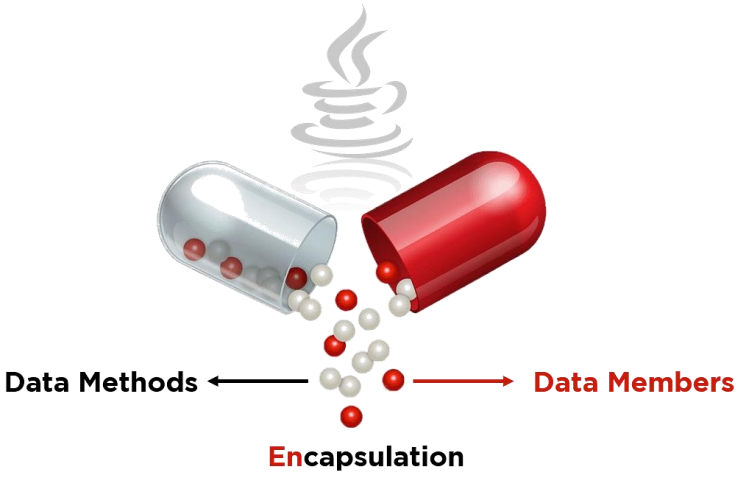
- Suppose you define the `Student` class and declare name and score as follows
  - When using, users can assign data to fields arbitrarily
  - What if valid scores are only from 0 to 10

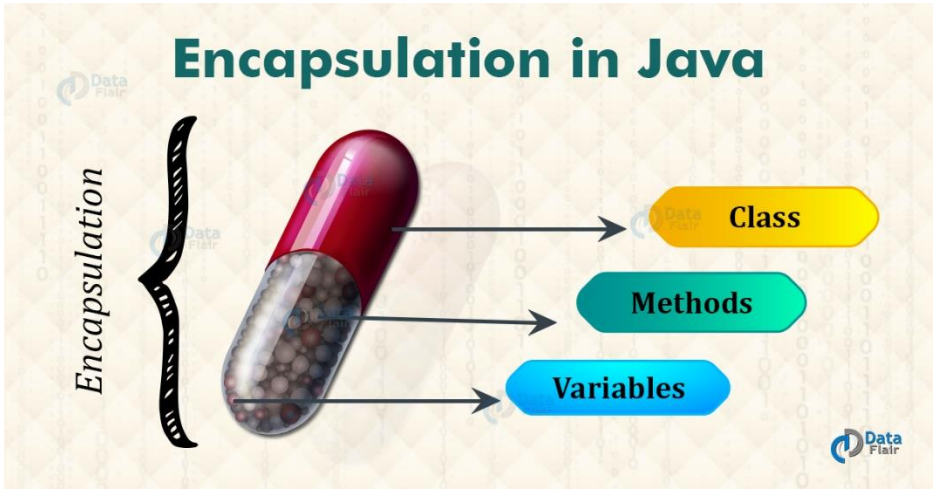
```
public class Student{
    public String name;
    public double score;
}
```

```
public class MyClass{
    public static void main(String[] args) {
        Student s = new Student();
        s.name = "Emma Watson";
        s.score = 20.5;
}}
```

Encapsulation

- Encapsulation is obfuscation in object orientation
  - Data fields should be hidden
  - Use methods to retrieve data fields
  - Purpose of concealment
  - Data protection
  - Enhance scalability





Encapsulation

- To hide information, use `private` for data fields

```
private double score;
```

- Add getter and setter methods to read and write private fields

```
public void setScore(double score) {
    this.score = score;
}
public String getScore(){
    return this.score;
}
```

Encapsulation

```
public class Student {
    private String name;
    private double score;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    public void setScore(double score) {
        if (score < 0 || > 10) {
            System.out.println("Score invalid");
        } else {
            this.score = score;
        }
    }
    public double getScore() {
        return this.score;
    }
}
```

Just add code to the `setScore()` method to handle invalid data

Encapsulation

```
public class MyClass {
   public static void main(String[] args) {
      Student sv = new Student();
      sv.setName("Emma Watson");
      sv.setScore(8.5);
   }
}
```

Method `toString()`

- A class should contain a public `toString()` method
- toString() method: Provides a string description of the class instance
- Invocation: `anInstanceName.toString()`
- Implicit invocation: Through `println()` or the `+` operator
- `println(anInstance)`: Implicitly triggers the `toString()` method for that instance

Method `toString()`

```
// return a String description of this instance
public String toString() {
    return "Name: " + name + ", score: " + score;
}
```

```
public class MyClass{
    public static void main(String[] args){
        Student sv = new Student();
        sv.setName("Emma Watson");
        sv.setScore(8.5);
        System.out.println(sv.toString());
        // Name: Emma Watson, score: 8.5
    }
}
```

Lecture summary

- Member Variables
- Member Methods
- Overloading
- Constructors
- Package
- The Access Control Modifiers
- Encapsulation
- Method `toString()`