Lab 8 – Object-Oriented Programming Concept (Episode II): Access Level and Encapsulation

Objectives:

- Understand Java access control
- Understand the concept of encapsulation
- Use encapsulation to protect data
- Practice writing Java program

Access Control Modifiers

The Java programming language provides access control mechanisms for controlling the accessibility/visibility of the class members. Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are three access modifier keywords, public, private, protected and four access control levels:

- public
- private
- protected
- package-private (no explicit modifier)

Class members with access modifier public are accessible by any class.

Class members with access modifier private are accessible inside the same class only. Other class cannot access them.

Class members with access modifier protected are accessible by any class or subclass within the same package. Classes are considered to be in the same package if they are in the same folder or directory.

Class members without any access modifier have package-private access level. Package-private members are accessible any class within the same package.

Table 1 summarizes classes and their accessibility to different access modifiers.

Table 1 - Access modifiers and accessibility

Class/ have access to	`public`	`protected`	*package-private (default, no modifier)*	`private`
Same class	Yes	Yes	Yes	Yes
Class – same package	Yes	Yes	Yes	No
Subclass – same package	Yes	Yes	Yes	No
Subclass – another package	Yes	Yes	Yes/No	No

Class – another package	Yes	No	No	No

Class modifiers at the top level can only have either public, or package-private (no explicit modifier).

At the top level, a class may be declared with the modifier `public`, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes as described in Lab 7.)

Class members can have all modifiers, public, private, protected, and no modifier (package-private).

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Let's look at a collection of classes and see how access levels affect visibility. Figure 1 shows the four classes in this example and how they are related.

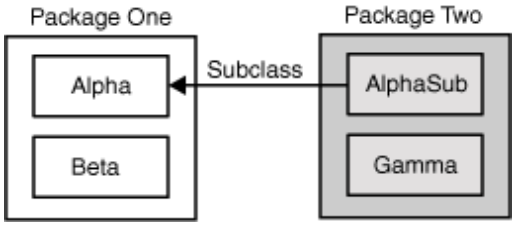


Figure 1 Classes and Packages of The Example Used to Illustrate Access Levels

Table 2 shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

Table 2 Visibility

Classes/Modifier	public	protected	no modifier	private
Alpha	Yes	Yes	Yes	Yes
Beta	Yes	Yes	Yes	No
AlphaSub	Yes	Yes	No	No
Gamma	Yes	No	No	No

Tips on Choosing an Access Level: If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

- Use the most restrictive access level that makes sense for a particular member. Use `private` unless you have a good reason not to.
- Avoid `public` fields except for constants. Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Your turn ①

Complete the following exercise.

1. Create a new project called `lab8`.
2. Create a new packaged called `one`.
3. Create a new class, `Alpha` in package `one`.

```
// Alpha.java
public class Alpha {
    public int pub;
    protected int prot;
    int pack;
    private int pri;

    public void alphaMethod() {
      Alpha alphaObj = new Alpha();
      System.out.println(alphaObj.pub);
      System.out.println(alphaObj.prot);
      System.out.println(alphaObj.pack);
      System.out.println(alphaObj.pri);
    }
}
```

Take a look in the method `alphaMethod()`. This method creates an instance of `Alpha` and try to print out each instance variable. There is no error in the program because the method is in the class `Alpha`, so it has access to all members of its own class.

4. Create a new class, `Beta` in `package1`.

```
// Beta.java
public class Beta {
    public static void betaMethod() {
      Alpha alphaObj = new Alpha();
      System.out.println(alphaObj.pub);
      System.out.println(alphaObj.prot);
      System.out.println(alphaObj.pack);
      System.out.println(alphaObj.pri);
    }
}
```

Do you see the error? There is an error at the statement `System.out.println(alphaOjb.pri)` since it tries to access the private member of an `Alpha` instance.

5. Create new package called `two`.
6. Create a new class called `AlphaSub` which extends `one.Alpha` in package `two`.

```
// AlphaSub.java
public class AlphaSub extends Alpha {
    public void alphaSubMethod() {
      Alpha alphaObj = new Alpha();
      System.out.println(alphaObj.pub);
      System.out.println(alphaObj.prot);
      System.out.println(alphaObj.pack);
      System.out.println(alphaObj.pri);

      System.out.println(this.prot);
    }
}
```

There are three errors.

a. The first error is `System.out.println(alphaObj.prot)` because when you access through an object (`alphaObject`), it does not access using inheritance mechanism. The rule for access is the *package-private*, since `AlphaSub` is in different package with `Alpha`.
b. The second error is `System.out.println(alphaObj.pack)` because it is in different package.
c. The third error is `System.out.println(alphaObj.pri)` because it try to access the private member.

Look at the statement `System.out.println(this.prot)`. `this.prot` accesses the protected member of `Alpha`. `AlphaSub` is a subclass of `Alpha`. From the access rule, `AlphaSub` can access protected member of `Alpha`.

7. Create a new class called `Gamma` in package `two`.

```
// Gamma.java
public class Gamma {
    public void alphaSubMethod() {
      Alpha alphaObj = new Alpha();
      System.out.println(alphaObj.pub);
      System.out.println(alphaObj.prot);
      System.out.println(alphaObj.pack);
      System.out.println(alphaObj.pri);

      System.out.println(this.prot);
```

```
    }
}
```

You can see that there are four errors. The first three errors have the same reasons as in `AlphaSub`. The forth error is because `Gamma` does not extends `Alpha`. There is no data member called `prot`.

Encapsulation

In object-oriented programming, the term *encapsulation* refers to the hiding of data within a class (a safe "capsule") and making it available only through certain methods. *Encapsulation* is important because it makes it easier for other programmers to use your classes and protects certain data within a class from being modified inappropriately.

Your turn

Exercise 2

1. Create a new package called `encapsulation`.
2. Create a new class called `PublicElevator` in package `encapsulation`.

```
// PublicElevator.java
public class PublicElevator {
    public boolean doorOpen = false;
    public int currentFloor = 1;
    public int weight = 0;

    public final int CAPACITY = 1000;
    public final int TOP_FLOOR = 5;
    public final int BOTTOM_FLOOR = 1;
}
```

The `publicElevator` declares all of its attributes to `public`, which permits their values to be changed without any error checking.

3. Create a new application called `PublicElevatorTest` in package `encapsulation`.

```
// PublicElevatorTest.java
public class PublicElevatorTest {

    public static void main(String[] args) {
        PublicElevator pubElevator = new PublicElevator();

        pubElevator.doorOpen = true;  // passengers get on
```

```
        pubElevator.doorOpen = false; // doors close

        // go down to floor 0 (below bottom of building)
        pubElevator.currentFloor--;
        pubElevator.currentFloor++;

        // jump to floor 7 (only 5 floors in building)
        pubElevator.currentFloor = 7;

        pubElevator.doorOpen = true;  // passengers get on/off
        pubElevator.doorOpen = false;
        pubElevator.currentFloor = 1; // go to the first floor
        pubElevator.doorOpen = true;  // passengers get on/off
        pubElevator.currentFloor++;   // elevator moves w/ door open
        pubElevator.doorOpen = false;
        pubElevator.currentFloor--;
        pubElevator.currentFloor--;
    }
}
```

Because the `PublicElevator` class does not use encapsulation, the `PublicElevatorTest` class can change the values of its attributes freely and in many undesirable ways. For example, on statement after `// go down to floor 0`, which might not be a valid floor. Also, on the statement, `pubElevator.currentFloor = 7`, the `currentFloor` attribute is set to `7` that, according to the `TOP_FLOOR` constant, is an invalid floor (there are only five floors).

> **Note** – Generally, you should use the `public` modifier only on methods and attribute variables that you want to be accessed directly by other objects.

The `private` modifier allows objects of a given class, their attributes, and operations to be inaccessible by other objects.

4. Create a new class called `PrivateElevator1` in package `encapsulation`.

```
// PrivateElevator1.java
public class PrivateElevator1 {
    private boolean doorOpen = false;
    private int currentFloor = 1;
    private int weight = 0;

    private final int CAPACITY = 1000;
    private final int TOP_FLOOR = 5;
    private final int BOTTOM_FLOOR = 1;
```

```
}
```

5. Create a new application called `PrivateElevator1Test` in package `encapsulation`.

```
// PrivateElevator1Test.java
public class PrivateElevator1Test {
    public static void main(String[] args) {
        PrivateElevator1 priElevator = new PrivateElevator1();

        /*
         * The following lines of code will not compile
         * because they attempt to access private variables.
         */

        priElevator.doorOpen = true;  // passengers get on
        priElevator.doorOpen = false; // doors close

        // go down to floor 0 (below bottom of building)
        priElevator.currentFloor--;
        priElevator.currentFloor++;

        // jump to floor 7 (only 5 floors in building)
        priElevator.currentFloor = 7;

        priElevator.doorOpen = true;  // passengers get on/off
        priElevator.doorOpen = false;
        priElevator.currentFloor = 1; // go to the first floor
        priElevator.doorOpen = true;  // passengers get on/off
        priElevator.currentFloor++;   // elevator moves w/ door open
        priElevator.doorOpen = false;
        priElevator.currentFloor--;
        priElevator.currentFloor--;
    }
}
```

The code does not compiled because the main method in the `PrivateElevator1Test` class is attempting to change the value of `private` attributes in the `PrivateElevator1` class.

The `PrivateElevator1` class is not very useful, however, because there is no way to modify the values of the class.

In an ideal program, most or all the attributes of a class are kept `private`. Private attributes cannot be modified or viewed directly by classes outside their own class, they can only be modified or viewed by methods of that class. These methods should contain code and business logic to make sure that inappropriate values are not assigned to the variable for an attribute.

6. Create a new class called `PrivateElevator2` in package `encapsulation`.

```
// PrivateElevator2.java
public class PrivateElevator2 {
    private boolean doorOpen = false;
    private int currentFloor = 1;
    private int weight = 0;

    private final int CAPACITY = 1000;
    private final int TOP_FLOOR = 5;
    private final int BOTTOM_FLOOR = 1;

    public void openDoor() {
        doorOpen = true;
    }

    public void closeDoor() {
        calculateCapacity();
        if (weight <= CAPACITY) {
            doorOpen = false;
        } else {
            System.out.println("The elevator has exceeded capacity.");
            System.out.println("Doors will remain open until someone exits!");
        }
    }

    // random weight for simulation
    private void calculateCapacity() {
        weight = (int)(Math.random() * 1500);
        System.out.println("The weight is " + weight);
    }

    public void goUp() {
        if (!doorOpen) {
            if (currentFloor < TOP_FLOOR) {
                currentFloor++;
                 System.out.println(currentFloor);
            } else {
                 System.out.println("Already on top floor.");
            }
        } else {
             System.out.println("Doors still open!");
        }
    }
```

```
    public void goDown() {
        if (!doorOpen) {
        if (currentFloor > BOTTOM_FLOOR) {
            currentFloor--;
            System.out.println(currentFloor);
        } else {
             System.out.println("Already on bottom floor.");
        }
        } else {
             System.out.println("Doors still open!");
        }
    }

    public void setFloor(int desiredFloor) {
        if ((desiredFloor >= BOTTOM_FLOOR) &&
            (desiredFloor <= TOP_FLOOR)) {
            while (currentFloor != desiredFloor) {
                if (currentFloor < desiredFloor) {
                    goUp();
                 } else {
                      goDown();
                 }
             }
        } else {
            System.out.println("Invalid Floor");
        }
    }

    public int getFloor() {
        return currentFloor;
    }

    public boolean getDoorStatus() {
        return doorOpen;
    }
}
```

7. Create a new application called `PrivateElevator2Test` in package `encapsulation`.

```
// PrivateElevator2Test.java
public class PrivateElevator2Test {

    public static void main(String[] args) {
        PrivateElevator2 privElevator = new PrivateElevator2();

        privElevator.openDoor();
        privElevator.closeDoor();
        privElevator.goDown();
```

```
        privElevator.goUp();
        privElevator.goUp();
        privElevator.openDoor();
        privElevator.closeDoor();
        privElevator.getClass();
        privElevator.openDoor();
        privElevator.goDown();
        privElevator.closeDoor();
        privElevator.goDown();
        privElevator.goDown();

        int curFloor = privElevator.getFloor();
        if (curFloor != 5 && !privElevator.getDoorStatus()) {
            privElevator.setFloor(5);
        }

        privElevator.setFloor(10);
        privElevator.openDoor();
    }
}
```

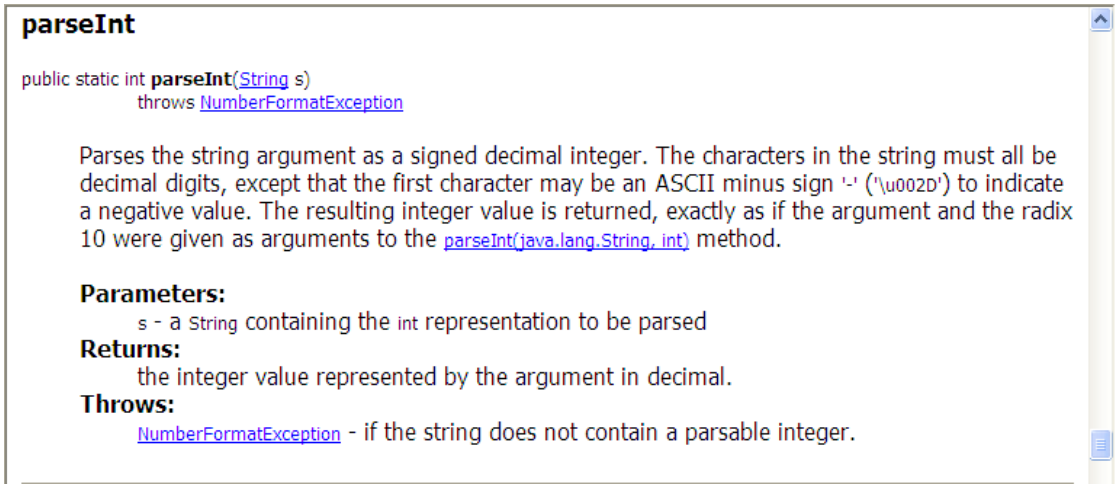
Because the `PrivateElevator2` class does not allow direct manipulation of the attributes of the class, the `PrivateElevator2Test` class can only invoke methods to act on the attribute variables of the class. These methods perform checks to verify that the correct values are used before completing a task, ensuring that the elevator does not do anything unexpected.

All of the complex logic in this program is encapsulated within the `public` method of the `PrivateElevator2` class. The code in the test class is, therefore, easy to read and maintain. This concept is one of the many benefits of encapsulation.

Your turn

Exercise 3

Consider the Java API document for the method `parseInt` of class `Integer`.



parseInt

public static int **parseInt**(String s)
throws NumberFormatException

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the parseInt(java.lang.String, int) method.

Parameters:
s - a String containing the int representation to be parsed
Returns:
the integer value represented by the argument in decimal.
Throws:
NumberFormatException - if the string does not contain a parsable integer.

You are to write your own Java program that will do the similar thing to Integer.parseInt.

1. Create a new package called `util`.
2. Create a new class called `Utilitiy`.
3. In `Utility.java`, add a new static method named `stringToInt` which has the following specification:

```
public static int stringToInt(String s)
            throws NumberFormatException
```

This method behaves like `Integer.parseInt(String)` which parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character my be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value.

Parameters:

s – a `String` containing the int representation to be converted

Returns:

the integer value represented by the argument in decimal.

Throws:

`NumberFormatException` – if the string cannot be converted to an integer.

4. Create a JUnit test for this class and method. In testStringToInt(), you must test for the following conditions:

value of s	*expected return value*
"0"	0
"xxx" where xxx is a string representing a positive integer <= Integer.MAX_VALUE	a positive integer xxx
"-xxx" where –xxx is a string representing a negative integer > Integer.MIN_VALUE	a negative integer –xxx
"xyz" where xyz connot be convert to an integer	throws NumberFormatException

5. Implement the stringToInt method so it passes all test cases.

References:

The Java Language Specification, Third Edition, http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html.

Controlling Access to Members of a class, *The Java Tutorial*, http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html

Lab 8 – Object-Oriented Programming Concept (Episode II): Access Level and Encapsulation

Task	Description	Result	Note
1	Access Level		
2	Encapsulation		
3	stringToInt		
4			
5			
6			
7			