Lecture 25
Visibility Modifiers/Specifiers

• Visibility modifiers can be used to specify the visibility of a class and its members.
• You can use the public visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as package-private or package-access.
• In addition to the public and default visibility modifiers, Java provides the private and protected modifiers for class members. This section introduces the private modifier. The protected modifier will be introduced in Section 11.13, The protected Data and Methods. The private modifier makes methods and data fields accessible only from within its own class.
package p1;

public class C1 {
    public int x;
    int y;
    private int z;
    
    public void m1() {
    }
    void m2() {
    }
    private void m3() {
    }
}

package p1;

public class C2 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        can access o.y;
        cannot access o.z;
        
        can invoke o.m1();
        can invoke o.m2();
        cannot invoke o.m3();
    }
}

package p2;

public class C3 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        cannot access o.y;
        cannot access o.z;
        
        can invoke o.m1();
        cannot invoke o.m2();
        cannot invoke o.m3();
    }
}

package p1;

class C1 {
    ...
}

package p1;

public class C2 {
    
    can access C1
}

package p2;

public class C3 {
    
    cannot access C1;
    can access C2;
}

A visibility modifier specifies how data fields and methods in a class can be accessed from outside the class. There is no restriction on accessing data fields and methods from inside the class.
Caution
The **private** modifier applies only to the members of a class. The **public** modifier can apply to a class or members of a class. Using the modifiers **public** and **private** on local variables would cause a compile error.

Note
In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a **private constructor**. For example, there is no reason to create an instance from the **Math** class, because all of its data fields and methods are static. To prevent the user from creating objects from the **Math** class, the constructor in **java.lang.Math** is defined as follows:

```java
private Math() {
}
```
Data Field Encapsulation

- To prevent direct modifications of data fields, you should declare the data fields private, using the private modifier. This is known as data field encapsulation.

- A private data field cannot be accessed by an object from outside the class that defines the private field. However, a client often needs to retrieve and modify a data field. To make a private data field accessible, provide a get method to return its value. To enable a private data field to be updated, provide a set method to set a new value.
Note
Colloquially, a **get** method is referred to as a *getter* (or *accessor*), and a **set** method is referred to as a *setter* (or *mutator*).

A **get** method has the following signature:

```java
public returnType getPropertyUserName()
```

If the **returnType** is **boolean**, the **get** method should be defined as follows by convention:

```java
public boolean isPropertyName()
```

A **set** method has the following signature:

```java
public void setPropertyName(dataType propertyValue)
```
Passing Objects to Methods

• Passing an object to a method is to pass the reference of the object.

You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. The following code passes the `myCircle` object as an argument to the `printCircle` method:

```java
public class Test {
    public static void main(String[] args) {
        // CircleWithPrivateDataFields is defined in Listing 8.9
        CircleWithPrivateDataFields myCircle = new CircleWithPrivateDataFields(5.0);
        printCircle(myCircle);
    }

    public static void printCircle(CircleWithPrivateDataFields c) {
        System.out.println("The area of the circle of radius "+ c.getRadius() + " is " + c.getArea());
    }
}
```

Java uses exactly one mode of passing arguments: pass-by-value. In the preceding code, the value of `myCircle` is passed to the `printCircle` method. This value is a reference to a `Circle` object.
Array of Objects

Chapter 6, Single-Dimensional Arrays, described how to create arrays of primitive type elements. You can also create arrays of objects. For example, the following statement declares and creates an array of ten Circle objects:

```java
Circle[] circleArray = new Circle[10];
```

To initialize `circleArray`, you can use a `for` loop like this one:

```java
for (int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle();
}
```

An array of objects is actually an array of reference variables. So, invoking `circleArray[1].getArea()` involves two levels of referencing, as shown in Figure 8.19. `circleArray` references the entire array; `circleArray[1]` references a Circle object.

**Note**

When an array of objects is created using the `new` operator, each element in the array is a reference variable with a default value of `null`. 

![Diagram](image-url)
Shadowing a *class* variable

- We can refer to a *class variable* defined inside the *same class* without using the *class name*. Example:

```java
public class ClassVar2 {
    public static double a; // <----- Class variable

    public static void main(String[] args) {
        // Body of method "main"
        a = 3.1415; // We can omit the classname in this method
        System.out.println(a);
    }
}
```

- Shadowing a class variable:

  - When the method has a *local variable* or a *parameter variable* defined in a scope whose *name* is **equal** to the name of the class variable, then:

    - The class variable with the same name can no longer accessible with the short hand notation in that scope !!!

- Example:

```java
public class ClassVar4 {
    public static double a = 3.1415;

    public static void main(String[] args) {
        boolean a = true; // Class var a new "shadowed"
        System.out.println(a); // prints true
        System.out.println(ClassVar4.a); // prints 3.1415
    }
}
```
Shadowing *instance* variables

- Recall that we can *omit* the *implicit parameter* `this` to access an *instance variable*

  (This is the *short hand notation* for instance variables)

- Shadowing can *also* occur with *instance variable* when a *parameter variable* or a *local variable* is defined inside an instance method that has the *same name* as an *instance variable*

Example:

```
public class BankAccount {
    public int accNum;    //
    public String name;   //
    public double balance; // Shadowed !

    /*
    * toString(): return a String containing
    * information of BankAccount
    */
    public String toString() {
        return "Account number: " + accNum
                 + " Name: " + name
                 + " Balance: " + balance;
    }

    /*
    * deposit(balance): Add "parameter balance" to balance
    */
    public void deposit(double balance) {
        balance += balance; // Add parameter balance to (this.)balance
    }

    /*
    * withdraw(balance): Subtract "parameter balance" from balance
    */
    public void withdraw(double balance) {
        if (balance >= balance)
            balance -= balance; // Subtract parameter balance from (this.)balance
    }
}
```
Problem:

- The same variable name balance is used to refer to 2 different variables:
  - The parameter variable balance
  - The instance variable (this.)balance

- The Java variable referencing rule (see: click here) will associate the variable name balance with the parameter variable balance.

Note:

- The Java compiler will not report any error because the statement is syntactically correct.
Overcome the *shadowing* problem with instance variables

- Use the full name to access the instance variable:
  
  `this.instanceVariableName`

Example:

```java
public class BankAccount {
    public int accNum;
    public String name;
    public double balance;  // Shadowed!

    public String toString() {
        return "Account number: " + accNum
            + ", Name: " + name
            + " Balance: " + balance;
    }

    public void deposit(double balance) {
        this.balance += balance; // Add parameter balance to this.balance
    }

    public void withdraw(double balance) {
        if (this.balance >= balance)
            this.balance -= balance; // Subtract parameter balance from this.balance
    }
}
```
Java provides an **automatic conversion feature** when the **+ operator** is used between:

- a *number typed data* and a *string typed data*

Java's **automatic** conversion rule for \( \text{number} \Rightarrow \text{string} \):

- When the **+ operator** is used between
  - a *number* and
  - a *string*

Then:

- the *number* is **automatically converted** to a *string*
- The **+ operator** is then **applied on 2 strings** (i.e., the **+ operator** is a **concatenation**)!
Automatic conversion of variables of a user-defined type to String

- Automatic conversion of variables of a user-defined type to string
  - Java also provides an automatic conversion feature when the `+` operator is used between:
    - a user-defined type variable (= object) and a string typed data

Java's automatic conversion rule for object → string:

- When the `+` operator is used between:
  - a object reference variable and
  - a string

Then:
- the object referred to by the object reference variable is automatically converted to a string
- The `+` operator is then applied on 2 strings (i.e., the `+` operator will be a string concatenation)

Example:

```java
public class Class10 {
    public static void main(String[] args) {
        String x;
        BankAccount stat1 = new BankAccount(123, "John", 1000);
        x = "Test --- " + stat1; // First converts stat1 to a String
        System.out.println(x);
    }
}
```

Output:
```
Test --- BankAccount$130019b
```

The string BankAccount$130019b is the result of the conversion of the object stat1 into the String type
Java's internal *mechanism* used to provide automatic object ⇒ String conversion

- The **automatic conversion** from an **object type** to the **string type** is accomplished by
  
  A rule that is **built-in** into the Java compiler !!!

- **Translating an expression** containing the **+ operator** by the Java compiler:
  
  The expression:
  
  ```java
  String + Object
  or
  Object + String
  ```

  is **translated** into:
  
  ```java
  String + Object  ⇒  String + Object.toString()
  or
  Object + String  ⇒  Object.toString() + String
  ```

Furthermore:

- **Every user-defined type** (= **object**) contains the method
  
  ```java
  public String toString()
  {
      ....
  }
  ```

- **Advanced Java programming note:**
  
  The `toString()` method is **inherited** from the **Super class Object**
public class Class10 {
    public static void main(String[] args) {
        String x;
        BankAccount stu1 = new BankAccount(123, "John", 1000);
        x = "Test --- " + stu1;  // Converted to: "Test --- " + stu1.toString()
        System.out.println(x);
    }
}

Explanation:

- The relevant statement is:
  
  \[
  x = "Test --- " + stu1;  // String + (BankAccount) Object
  \]

- Due to the automatic string conversion rule, the Java compiler will translate this statement into:
  
  \[
  x = "Test --- " + stu1.toString();  // use "toString()" to convert object to String
  \]

- Our `toString()` method defined inside the `BankAccount` class is now invoked, which returns the string:

  Account number: 123, Name: John, Balance: 1000.0

This returned string is then concatenated with "Test --- ".