Class organization and access control

Modifiers

Modifiers are keywords that you add to the definition of classes, methods and variables in order to change their meaning. The Java language supports the following modifiers:

- Modifiers for controlling access to a class, method, or variable: **public**, **protected**, and **private**
- The **static** modifier for creating class methods and variables
- The **final** modifier for finalizing the implementations of classes, methods, and variables
- The **abstract** modifier for creating abstract classes and methods
- The **synchronized** and **volatile** modifiers, which are used for threads

Modifiers

Examples:

```
public class Calc extends javax.swing.JApplet {
    // …
}
private boolean offline;
static final double weeks = 9.5;
protected static final int MEANING_OF_LIFE = 42;
public static void main(String[] arguments) {
    // body of method
}
```

Access control modifiers

- Determine which variables and methods of a class are visible to other classes.
- Java defines four access levels: **public**, **private**, **protected** and **default**.

Access control modifiers: default access

A variable or method declared **without any access control modifier** is available to any other class in the same package.

Examples:

```
String version = “0.7a”;
boolean processOrder() {
    return true;
}
```

Access control modifiers: private access

- The **private access control modifier** access hides a method or a variable from being used by any other class. The only place these methods or variables can be accessed from is from within their own class.
- Useful when:
  - Other classes have no reason to use that variable
  - Another class could wreak havoc by changing the variable in an inappropriate way

Access control modifiers: private access

- Example:

```
class Logger {
    private String format;

    public String getFormat() {
        return this.format;
    }

    public void setFormat(String format) {
        if ( (format.equals(“common”)) || (format.equals(“combined”)) ) {
            this.format = format;
        }
    }
}
```

- Using the private modifier is how Java makes the **encapsulation** OOP principle possible.

Access control modifiers: public access

Public methods and variables are available to any class that wants to use them.

Example:

```
public static void main(String[] args) {
    // ...
}
```

Access control modifiers: protected access

The **protected modifier** limits the access to a method or a variable to the following two groups:

- Subclasses of a class
- Other classes in the same package

Example:

```
class AudioPlayer {
    protected boolean openSpeaker(Speaker sp) {
        // implementation details
    }
}

class StreamingAudioPlayer extends AudioPlayer {
    protected boolean openSpeaker(Speaker sp) {
        // implementation details
    }
}
```

Access control modifiers summary

Visibility	public	protected	default	private
From the same class	yes	yes	yes	yes
From any class in the same package	yes	yes	yes	no
From any class outside the same package	yes	no	no	no
From a subclass in the same package	yes	yes	yes	no
From a subclass outside the same package	yes	yes	no	no

Access control and inheritance

When creating a subclass and overriding a method, the new method can't be made more restrictively controlled than the original but can be made more public. The following rules apply:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private or with default access.
- Methods with default access control (no modifier was used) can retain their access control or be declared protected or public in a subclass.
- Methods declared private are not inherited at all, so the rules don’t apply.

Accessor and mutator methods

Fields guarded by the access control modifiers should sometimes be used by other classes. This can be achieved using **accessor** and **mutator** methods.

- Accessor methods can be used to get access to a "hidden" field.
- Mutator methods can be used to achieve controlled modification of fields.

Example:

```
class Logger {
    private String format;

    public String getFormat() { return this.format; }

    public void setFormat(String format) {
        if ( (format.equals(“common”)) || (format.equals(“combined”)) ) {
            this.format = format;
        }
    }
}
```

Final classes, methods and variables

- The **final** modifier can be used to indicate that a class, method or variable should not be "changed".
- In particular:
  - A final class cannot be subclassed.
  - A final method cannot be overridden by any subclasses.
  - A final variable cannot change in value.

Final variables

Final variables are often called **constants** because they do not change their value over time.

Examples:

- `public static final int TOUCHDOWN = 6;`
- `final String title = “Captain”;`

Final methods

Final methods can't be overridden by a subclass.

Example:

```
public final void getSignature() {
    // body of method
}
```

Final classes

- A final class can't be subclassed by another class

  Example:

```
public final class ChatServer {
    // body of method
}
```

- Final classes in the JRE: java.lang.String, java.lang.Math, java.net.URL, etc.

Abstract classes and methods

- Some of the classes in a hierarchy don't need to be instantiated directly. Instead, they serve as a place to hold common behavior and attributes shared by their subclasses. These classes are called **abstract classes**.
- Abstract classes can contain:
  - Anything that a normal class contains (fields, constructors, methods, etc.)
  - Abstract methods - methods with signature but no implementation
- Example:

```
public abstract class Vehicle {
    public abstract void honk();
}
```

Packages

- Packages are a way of organizing classes. It's usually common to group under the same package (and sub packages) those classes which are related in terms of their functionality.
- Advantages of using packages:
  - They enable the organization of classes into units similar like folders and directories on the hard drive.
  - They reduce the problem of conflicting names.
  - They enable protecting classes through access control modifiers.

Using packages

In order to use a class contained in another package, the following techniques can be used:

- If the class is from the "java.lang" package, it can be used directly as the classes from this package are automatically available.
- If the class is from another package, it can be referred to by its full name.

  Example:

  ```
  java.awt.Font text = new java.awt.Font()
  ```

- If the class is from another package and is used often, it can be imported individually or the entire package can be imported.

  Examples:

  - `import java.util.Vector;`
  - `import java.awt.*;`

Class name conflicts

In case two classes from different packages need to be used at the same time, both can't be used by importing their package. There are two solutions:

- Refer to both classes through their full names.
- Import the package of one of the classes and refer to the second one through its full name.

Example:

```
import java.sql.*;
// …
Date now = new Date();
java.util.Date = new java.util.Date();
```

Inner classes

- Inner classes are defined inside other classes as if they were methods or fields.
- Advantages:
  - Inner classes are invisible to all other classes, which means that you don’t have to worry about name conflicts between it and other classes.
  - Inner classes can have access to variables and methods within the scope of a top-level class that they would not have as a separate class.

Example: SquareTool

Exercises

Exercise: ZipCode

Create a ZipCode class that uses access control to ensure that its zipCode instance variable always has a five or nine-digit value.