Software Development (CS2500)

Lecture 11: Writing Classes

M.R.C. van Dongen

October 22, 2010

Contents

1 Introduction

This lecture corresponds to the start of Chapter 4 but the presentation is different. The main objectives are as follows.

- Discuss the structure and content of class definitions;
- Study the relationship of object state and instance data;
- Learn how to use visibility modifiers to protect methods and data;
- Examine the structure and purpose of constructors; and
- Write a class that represents a die.

2 Classes and Objects Revisited

Remember that a class is a blueprint of its *instances*. Here the instances of the class are the objects which are created by the class.

- The class represents the concept of the object.
- Each object created by the class is a realisation of the concept.
- Each object has as a state, which is defined by the object's attributes.
- The class establishes that each object has these attributes.
- Each object has behaviours. Here the behaviour are the instance methods of the class. Some behaviours may affect the object's state.
- The class defines the object's methods.

3 Anatomy of a Class

3.1 Variable and Method Declarations

Classes contain variable and method declarations. There are two kinds of declarations: class or instance declarations. Instance variables and instance variables are owned by objects. The word 'instance' stems from the fact that each object is an instance of the class. Class variables and class methods are owned by the class.

Instance variable: Instance variables define the *attributes* that are owned by the *objects*. For each attribute declaration in the class, each newly created object is given its own variable to represent the attribute's current value. If there are n instance variable declarations then each new object gets n different variables to represent their values. Different objects may have different values for the same attribute. Instance variables can only be accessed using an object reference:

'⟨`object reference`⟩.⟨`attribute`⟩'.

Class variable: Class variables define the *attributes* that are owned by the *class*. Each class variable declaration corresponds to a *single* variable. These variables can only be accessed using the class:

'⟨class name⟩.⟨attribute⟩'.

Class variables are different from instance variables. For example, changing the value of a class variable has a global effect for the whole class.

Instances method: Instance methods are methods which are owned by the *objects*. These methods can only be accessed using an object reference:

'⟨object reference⟩.⟨method⟩(⟨arguments⟩)'.

Instance methods can see all class variables and class methods of the current class. In addition they can see their object's instance variables and instance methods. Finally, each instance method has access to *its* object, which is the object which called the instance method. For convenience we shall refer to this object as 'the *current* object' or as 'this'. Inside the instance method the current object can only be accessed indirectly. To access the object, you use the Java keyword 'this', which acts as the object reference to the object. This is explained in the next subsection.

Class method: Class methods define methods which are owned by the *class*. Each class method declaration correspond to a single method which is owned by the defining class. These methods may be accessed anywhere using the class:

'⟨class name⟩.⟨method⟩(⟨arguments⟩)'.

However, inside the defining class you may omit the '⟨class name⟩.' part. Class methods only have access to the class attributes and class methods.

3.2 this

There are two techniques to access an object's instance variables and instance methods from within the object's instance methods. The techniques to access instance variables and methods are equivalent. However, one technique uses the Java keyword 'this', whereas the other technique does not. The method that uses this is more verbose and is not commonly used. However, there is a case where it is needed. This is explained further on. The following explains the difference between the two techniques.

with: When you write 'this' in a Java instance method, the 'this' works just as an object reference. Specifically, the value of 'this' is the reference to the object that called the method. For example, if ⟨reference⟩ is a reference to the object that called the method and if '⟨reference⟩.⟨method⟩(⟨arguments⟩)' instructed the object to carry out the method call, then the value of 'this' inside the method is the same as that of ⟨reference⟩. So this and ⟨reference⟩ refer to the same object. In short 'this' corresponds to the "current" object. Since this is an object reference, writing 'this.⟨instance variable⟩' and 'this.⟨instance method⟩(⟨arguments⟩)' works as expected.

without: The more commonly used technique to access instance variables and instance methods inside instance methods does not use '`this.`', so you write '⟨`instance variable`⟩' or '⟨`instance method`⟩`(` ⟨`arguments`⟩ `)`'.

However, there is one case where you need the '`this.`' notation. This case arises if an instance method has a formal parameter or variable whose name shadows that of an instance variable. If ⟨`name`⟩ is the name of the formal parameter or variable, then ⟨`name`⟩ and '`this.` ⟨`name`⟩' are *different*. Specifically, ⟨`name`⟩ is the variable or parameter and '`this.` ⟨`name`⟩' is the instance variable. The following is an example.

Don't Try this at Home

```
public class DontDoThis {
    private final int value;

    public void possible( ) {
        int value = 1;  // grand.
    }

    public void alsoFine( int value ) {
        value = 1;      // grand.
    }

    public void impossible( int argument ) {
        value = 1;      // Noooooooooo!
    }
}
```

If you try to compile this you will get a compile-time error because `Java` won't let you assign a value to the `final` attribute `value` inside the method `impossible( )`.

In a similar vein, you sometimes need to use the '⟨`class name`⟩`.`' notation to distinguish between a class attribute and a variable or formal parameter inside a method.

Java

```
public class Example {
    private static final int value = 1;

    public static void example( ) {
        int value = 2;
        System.out.println( (2 * Example.value) + " = " + value );
    }
}
```

3.3 Basic Declarations

The following shows the basic form of instance variable and instance method declarations.

```Java
// Instance variable declaration:
⟨type⟩ ⟨attribute⟩;

// Instance method declaration:
⟨return type⟩ ⟨method⟩( ⟨formal parameter list⟩ ) {
    ⟨body⟩
}
```

By adding the modifier 'static' you can "turn" an instance variable declaration "into" a class variable declaration. Likewise, adding the modifier 'static' to an instance method declaration "makes" it a class method declaration.

```Java
// Class variable declaration:
static ⟨type⟩ ⟨attribute⟩;

// Class method declaration:
static ⟨return type⟩ ⟨method⟩( ⟨formal parameter list⟩ ) {
    ⟨body⟩
}
```

3.4 Example

In this section we shall implement a class called Die. A Die object represents a single die. The class defines instance methods which allow us to roll the die and get its current face value.

The following is the basic structure of the class. Usually the attributes are defined at the start of the class, but this is not a requirement. Remember that the name of the class is the same as the basename of the file it is in.

```Java
//*************************************************
// File:   Die.java
// Author: Java Joe
//
// Represents single die.
//*************************************************

⟨Import statements.⟩

public class Die {
    ⟨Variable declarations.⟩
    ⟨Method declarations.⟩
}
```

The following implements the variable declarations.

```Java
import java.util.Random;

public class Die {
    // Class variables:
    private static final Random random = new Random( );
    private static final int MAX_FACE_VALUE = 6;
    // Instance variable:
    private int faceValue; // Current face value.

    ⟨Method declarations.⟩
}
```

Attaching the modifier '`private`' to a variable or method declaration prohibits references to the variable or method from outside the class `Die`. This is good software engineering practice because it increases class robustness, flexibility, and extensibility. We shall see more about this later.

Notice that the primitive type variable '`MAX_FACE_VALUE`' is a constant. It is interesting to notice that it is spelled using upper case letters. This is a common convention among `Java` programmers, which makes it easier to recognise constant primitive type variables.

By initialising the class variables as part of the class variable declarations these variables are initialised when the `Die` class is created.

We shall now implement the methods. The following implements one of the most important methods: the *constructor* method. You use the constructor method each time you construct a new `Die` object. Constructing a new `Die` object is done with `new`. Writing '`new Die( )`' creates the new `Die` object and returns a unique reference to the object. The main purpose of the constructor method is to initialise the object's instance variables. In our case, there's only one: `faceValue`. `Java` primitive type valued numeric attributes which are not *explicitly* initialised are initialised to zero. Since zero is not a meaning face value for `Die` objects we shall assign `faceValue` an allowed value. Quite arbitrarily we assign it the value 1.

```Java
//-----------------------------------------------
// Constructor: Sets up the initial default face value.
//-----------------------------------------------
public Die( ) {
    faceValue = 1;
}
```

Note that we really should have used a constant variable to initialise the variable `faceValue` to its default value. For example, the following is much clearer because it doesn't use the *magic constant* 1 inside the method definition.

```Java
private static final int INITIAL_FACE_VALUE = 1;
//-----------------------------------------------
// Constructor: Sets up the initial default face value.
//-----------------------------------------------
public Die( ) {
    faceValue = INITIAL_FACE_VALUE;
}
```

You may also declare constructors with arguments. These are useful if you want to initialise the new object's attributes with provided values or, more generally, initialise them with values which are computed from the arguments.

The keyword '`this`' can also be used inside constructor methods. This is a common idiom if the name of one of the arguments of the constructors shadows that of an instance variable. The following demonstrates the technique.[1]

```Java
//-----------------------------------------------
// Constructor: Sets up the explicit face value.
//-----------------------------------------------
public Die( int faceValue ) {
    this.faceValue = faceValue;
}
```

The following implements the first instance method. The main purpose of this method is to roll the current object's die and save the resulting face value by assigning it to the object's instance variable `faceValue`.

```Java
//-----------------------------------------------
// Rolls the die.
//-----------------------------------------------
public void roll( ) {
    faceValue = 1 + random.nextInt( MAX_FACE_VALUE );
}
```

Note that the statement inside the instance method '`void roll( )`' refers to the instance variable `faceValue` of the "current" object: the object that called the method `roll( )`. Page 3 explained that there are two programming technique to access instance variables and instance methods inside instance methods. The technique of programming in the previous example does not use the keyword '`this`'. However, we could have also used the other technique. The following shows how this is done.

[1] For simplicity we assume that the argument of the constructor is a proper face value.

```Java
//-------------------------------------------------
// Rolls the die.
//-------------------------------------------------
public void roll( ) {
    this.faceValue = 1 + random.nextInt( MAX_FACE_VALUE );
}
```

This form is more verbose, not very common, but perfectly valid.

The following implements the next instance method. It is used to get the object's (the die's) current face value.

```Java
//-------------------------------------------------
// Returns current face value.
//-------------------------------------------------
public int getFaceValue( ) {
    return faceValue;
}
```

The following implements the last method in our class. The purpose of this method is to compute the current face value as a `String`.

```Java
//-------------------------------------------------
// Returns string representation of face value.
//-------------------------------------------------
@Override
public String toString( ) {
    return Integer.toString( faceValue );
}
```

Remember from Lecture 10 that each class is a subclass of the `Object` class. As such, each subclass inherits the methods that are defined in the `Object` class. One of these inherited methods is the instance method '`String toString( )`'. The inheritance mechanism allows subclasses to inherit a default behaviour by reusing the default implementation of their superclass. We've seen this mechanism in Lecture 6 where the `Square`, `Triangle`, and `Circle` classes inherited the default '`rotate( )`' and '`playSound( )`' behaviour. The inheritance mechanism is useful because there is no need to re-implement inherited behaviour.

For `Die` objects the default behaviour of the instance method '`String toString( )`' (as provided by the `Object` class) is not particularly useful. It would be much more meaningful if `toString( )` returned the object's current face value as a `String`. To accomplish this behaviour we have to *override* the method `toString`. *Overriding* the method means implementing a more specific (as opposed to the default) object behaviour.

To err is human. To make sure we really, really, really override the method '`String toString( )`' in our program we added the magic spell '`@Override`' just before the definition of the overriden method.

Adding the spell instructs the compiler to report an error if the overridden method is not known — such errors frequently happens as a result from typos. Without the magic spell such errors may go unnoticed for a long time. However, with the spell such errors are detected at compile time.

With the current implementation of our `Die` class we may use it as follows.

```Java
public class Casino {
    public static void main( String[] args ) {
        Die die1 = new Die( ); // construct new Die object.
        Die die2 = new Die( ); // construct another Die object.
        while (die1.getFaceValue( ) == die2.getFaceValue( )) {
            die1.roll( ); // Roll the first Die.
            die2.roll( ); // Roll the second Die.
        }
        // Print final face values.
        System.out.println( "Player 1 eventually got " + die1 + "." );
        System.out.println( "Player 2 eventually got " + die2 + "." );
        // Announce winner.
        int winner = die1.getFaceValue( ) < die2.getFaceValue( ) ? 2 : 1;
        System.out.println( "The winner is player " + winner + "." );
    }
}
```

It is important to notice that the two statements that print the final face values use the `Die` object reference variables `die1` and `die2` in combination with `String` concatenation. When `Java` sees an object reference, ⟨`reference`⟩, in a situation like this it substitutes '⟨`reference`⟩`.toString( )`' for ⟨`reference`⟩. Since we overrode the method `toString( )` in the `Die` class, this inserts the object's current face value, which is exactly what we need.

3.5 Constructors

Constructors do not have a return type. This is why they do not have a `return` statement. Constructors may also take arguments. `Java` automatically initialises instance variables. The default values are the same as for array entries. So `int` attributes are initialised to `0`, `double` attributes to `0.0`, `object` attributes to `null`, and so on. However, despite this default initialisation mechanism, it is considered good practice to explicitly initialise instance variables in a constructor method.

4 Encapsulation

Objects should be *self-governing*. This means their instance data should be modified by them and no other entities. Objects are *encapsulated* from the rest of the program: each object is a container (or capsule) for its instance variables and instance methods. They should only interact via a well-defined *interface*.

This may be accomplished by *data/method hiding*. Here *data/method hiding* is the ability to shield the data/method from external access.

In Java you implement data and method hiding with *visibility modifiers*. For the moment, there are two visibility modifiers: `private` and `public`. Table 1 describes the purpose of these visibility modifiers.

	`public`	**`private`**
Variables	Violates encapsulation	Enforces encapsulation
Methods	Service method	Support method

Table 1: Effects of `public` and `private` visibility.

5 Getters and Setters

Getter and *setter* methods get and set the values of instance variables. These methods are usually called `get`⟨`Name`⟩ and `set`⟨`Name`⟩, where ⟨`Name`⟩ is the name you get by capitalising the first letter of the name of the instance variable. The following is a short example.

```Java
private double percentage;

public double getPercentage( ) {
    return percentage;
}

public double setPercentage( double newPercentage ) {
    percentage = newPercentage;
}
```

The following are some advantages of using getter and setter methods.

- Restricting the getting and setting of variables with getters and carefully designed setters increases program robustness. For example, we may disallow attempts to assign impossible face values. We may handle assignment errors gracefully.
- In particular it helps maintain complex invariants about the object's instance variables. For example, in a bank transaction system, the moneys received by Client A from Client B should be equal to the moneys transferred from Client B to Client A.
- Information hiding also improves flexibility and extendibility. For example, client software that is unaware of the actual representation of the class cannot depend on the actual representation. By hiding the information, a change in the internal class representation cannot break client software.

6 Scope

The *scope* of a variable is where the variable is "visible" in the program. The scope of variables works as follows. We're restricting our attention to formal parameters, variables which are local to a method or block, and `public` and `private` class and instance variables.

parameters: Visible in the whole method.

local variables: Variables which are declared inside a method are visible from their declaration until the end of the smallest *block* they're in. Here a block starts with an opening brace (`{`) and ends with a closing brace (`}`).

`public` variables: Public class and instance variables are visible in any class. However, the scope excludes the scopes of formal method paramers or local variables with the same name.

`private` variables: Public class and instance variables are visible in the class they're in. However, the scope excludes the scopes of formal method parameters or local variables with the same name.

`Java` allows declarations anywhere inside a class/method/block. Redeclaring an instance variable as a local variable is only allowed in methods. ther variable redeclarations are not allowed.

The following is an example. For sake of the example, the declaration of the attribute `var` *follows* the definition of a method. This style of declaration should be discouraged because it is much clearer if all attribues are declared at the start of the class.

```java
public class Scope { // START OF BLOCK
    // turns on scope of attribute var.

    public void overlapping1( int var ) { // START OF BLOCK
        // turns on (off) scope of parameter (attribute) var.
        ... // parameter var
    }   // END OF BLOCK
        // turns off (on) scope of parameter (attribute) var.

    private int var;

    public void overlapping2( int param ) { // START OF BLOCK
        // turns on scope of parameter param.
        ...
        { // START OF BLOCK.
          ...
          int var;
          // turns on (off) scope of local variable (attribute) var.
          ...
        } // END OF BLOCK
          // turns off (on) scope of local variable (attribute) var.
    } // END OF BLOCK
        // turns off scope of parameter param.

    public void nonOverlapping( int par ) { // START OF BLOCK
        // turns on scope of parameter par
        ...
    }   // END OF BLOCK
        // turns off scope of parameter par
}
```

7 Lifetime

Variable *lifetime* is determined as follows:

Class variable: Class variables are created when their class is created. They cease to exist when their class is no longer needed.

Instance variable: The lifetime of an instance variable is the same as the lifetime of the object that owns the instance variable.

Local variable: The lifetime of a local variable is the same as the lifetime of the block that defines the variable's scope.

Parameter: The lifetime of the formal parameter of a method is the same as the duration of the (current) method call.

8 For Wednesday

Study the notes and read Pages 71–76.

9 Acknowledgements

The material presented in these notes is partially based on [Lewis and Loftus, 2009].

References

[Lewis and Loftus, 2009] John Lewis and William Loftus. *Java Software Solutions* Foundations of Program Design. Pearson International, 2009.