

Software Development (cs2500)

Lecture 12: Methods and Class Design

M.R.C. van Dongen

October 29, 2010

Contents

1	Introduction	1
2	Why Methods?	2
3	Pass-by-Value	2
3.1	Parameter Taxonomy	2
3.2	The Mechanism	4
4	Examples	4
5	Scanner Objects	5
6	A Ball Game	6
6.1	The Ball Class	7
6.2	The Hand Class	8
6.3	The main Method	10
7	For Monday	10

1 Introduction

The first part of this lecture corresponds to the start of Chapter 4 but the presentation is different. The last two sections are not in the book. The main objectives are as follows.

- Study methods and why they are useful.
- Learn the *pass-by-value rule*. This rule is important because it describes how Java methods should be evaluated.

- Simulate the evaluation of methods.
- Study the use of `Scanner` objects, which are commonly used for parsing input.
- Learn more about class design by carrying out a case study for a simple “game”. We shall learn that much of class design boils down to looking for nouns, verbs, predicates, and properties in the problem specification.

2 Why Methods?

Methods are arguably one of the most interesting things in computer science. In this section we shall study methods and why we should bother about them.

Methods are useful for several reasons:

- Methods are interfaces for parameterised computations. Using a single method call we can carry out a complicated well-defined computation.
- Method calls provide reusable computations. Methods allow us to implement an algorithm once, and use the algorithm several times. This aspect is the basis for, possibly tedious, automation.
- Method calls are the building blocks of more complex computations. An algorithm that is written in terms of a short sequence of method calls is easier to understand and easier to develop.
- Method calls are the only mechanism to change `private` variables.
- Methods encapsulate computations. They separate an algorithm’s implementation from where you use the algorithm. From a software engineering point of view, this is a big advantage. If the method is called in several locations in a program, then a *local* change in the implementation of the method will have a *global* impact on all method calls in the program. With a copy-and-paste approach changes would be required at *several* places in your program.

3 Pass-by-Value

Different programming languages have different mechanisms for evaluating methods. Java methods are evaluated using the *pass-by-value* paradigm. Before we can study pass-by-value we need to agree on some common vocabulary that allows us to explain the pass-by-value mechanism. The following section introduces the vocabulary. This is followed by the explanation of the mechanism.

3.1 Parameter Taxonomy

This section is a short introduction to *parameter* (or argument) taxonomy. Understanding it greatly helps us understand the difference between on the one hand the parameters in a method definition and on the other the parameters of a method call.

The following explains the difference:

Formal parameter: The method's *formal parameters* are the “variables” inside the method's parameter list in the method definition. The following shows the basic form of a method definition and its formal parameters.

```
Java
<visibility modifier> <static option>
<type> <method name>( <type>1 <formal parameter>1,
    ...,
    <type>n <formal parameter>n ) {
    <body>
}
```

Actual parameter: A method call's *actual parameters* are the arguments inside the method call's parameter list. The following shows the basic form of an instance method call and its actual parameters. Class method calls are written in a similar way but you leave out the ‘<reference>.’ part. In this example, it is assumed that the method does not return a value.

```
Java
<reference>.<method name>( <actual parameter>1,
    ...,
    <actual parameter>n );
```

It is important to understand that actual parameters may be literals, variables, as well as more general expressions.

```
Java
System.out.println( "The answer is " + 42 + "." );
```

Consider the following Java code.

```
Java
public static int f( int a, int b ) {
    return a + b;
}

public static void g( int c ) {
    int a = f( 1, 2 + c );
    int d = f( 1 + 3, a );
}
```

The formal parameters of the method `f` are `a` and `b`. The method `g` has only one formal parameter: it is `c`. There are two method calls and both are to `f`. Therefore there are four actual parameters and they are given by `1`, `2 + c`, `1 + 3`, and `a`.

3.2 The Mechanism

Having established the difference between formal and actual parameters we are now ready to study the pass-by-value evaluation mechanism. The following are the rules. To evaluate a method call with n parameters:

1. Create a fresh variable for each parameter. Each variable is used to represent the value of a formal parameter in the method call.
2. For i from 1 to n (from left to right):
 - (a) Evaluate the i th actual parameter.
 - (b) Assign the result of this evaluation to the i th fresh variable.
3. Use the fresh variables to represent the values of the formal parameters and carry out the statements in the method body.
4. If the method returns a result, then substitute the result for the method call.

Notice that we first carry out computations to evaluate the expressions that make up the actual parameters. When we're finished with these computations, we assign the results of these computations to the formal parameters. When doing this, we always assign a *copy* of the actual parameter's value. Even if the i th actual parameter is a variable $\langle \text{var} \rangle$ and the i th formal parameter also has the name $\langle \text{var} \rangle$, then no assignment to the formal parameter inside the method can affect the value of the actual parameter.

Having said that, instance methods can “see” instance variables, so it *is* possible to change the value of an instance variable as a result of a call to an instance method.

The temporary variables that are used to represent the actual parameters are stored on a first-in-last-out data structure which is called a *stack*.

- When the method is called, variables are created on top of the stack.
- Upon returning from the method, this space is removed from the stack.

The stack is also used to represent local variables in blocks.

- When the block is entered, variable are created on top of the stack.
- When control leaves the block, this space is removed from the stack.

4 Examples

At this stage, you are invited to go to today's presentation for some examples that show the pass-by-value mechanism. These examples simply could not be included in the notes: it would have been a waste of paper. You may find the presentation at <http://csweb.ucc.ie/~dongen/cs2500/10-11/12/Slides.pdf>.

5 Scanner Objects

This section is an introduction to Scanner objects, which are a useful tool for programs that require input. A Scanner is an object which can parse primitive types and strings using regular expressions. Reading input is really easy with Scanner objects.

A Scanner object breaks its input into tokens using a *delimiter* pattern. By default the delimiter is whitespace. The resulting tokens may then be converted into values of different types using the Scanner object's various next methods.

The following creates a Scanner object that lets you read from standard input.

```
Scanner scan = new Scanner( System.in );
```

Java

Having created the Scanner object, you can use it to read things.

```
String text = scan.next( ); // Get next string.  
int i = scan.nextInt( ); // Get next int.
```

Java

You may also create Scanner objects that let you read from Files and even from Strings. We shall postpone reading from Files until some other lecture. Reading from Strings is discussed further on.

The following are some of the methods provided by the Scanner class.

Scanner(InputStream source): This constructor creates a Scanner object from an InputStream, which is a primitive Java input object. The class variable in from the class System — it is called System.in — is an InputStream object reference variable which takes ints input from standard input.

Scanner(String str): This is another constructor which lets the resulting Scanner get its tokens from str.

void close(): This instance method *closes* the Scanner. Basically, this is what you do when you're finished reading with the Scanner. It is not allowed to read from a Scanner after calling its close() method.

boolean hasNext(): This method returns true if and only if the Scanner has another token on its input.

String next(): This method returns the Scanner next token. on its input. The method will work only if hasNext() is true. Otherwise, it will cause a run-time exception.

boolean hasNextInt(): This method returns true if and only if the next token on the Scanner's input is an int. There are also instance methods hasNextByte(), hasNextShort(), ..., which can be used to determine if the Scanner's next token is a byte, short,

int nextInt(): This method removes the next token from the Scanner’s input and returns it as an int. The method will work only if hasNextInt() is true. Otherwise, it will cause a run-time exception. There are also instance methods nextByte(), nextShort(), ..., which can be used to get the Scanner’s next a byte, short,

The following is a self-contained example, which outputs three lines. The first contains the text ‘one’, the second ‘two’, and the third ‘three’,

```
import java.util.Scanner;

public class Test {
    public static void main( String[] args ) {
        Scanner scan = new Scanner( "one two three" );
        while (scan.hasNext( )) {
            System.out.println( scan.next( ) );
        }
    }
}
```

The following is another example. This time the Scanner object reads from standard input. While there are tokens on the Scanner’s input the program will get the next token and print it.

```
Scanner scan = new Scanner( System.in );
while (scan.hasNext( )) {
    System.out.println( "Read: " + scan.next( ) );
}
scan.close( );
```

6 A Ball Game

In this section we shall implement a simple ball “game”. As part of the implementation of the game we shall learn about class design.

When you think about it, designing a Java for a complex application is not a science. For example, how do we choose the classes? Once we’ve decided on the classes, how do we choose the attributes, and how do we choose the methods? The case-study, which we shall carry out in a moment, shows that the problem specification provides many clues.

A common technique to find classes is to look for actors in the specification. This works, because the actors correspond to the objects. The actors do things (verbs): these are the methods. The actors own things, these are the attributes. Here “owning” things includes predicates. A predicate is a term designating a property or relation. For example, ‘being tall’, having ‘a length’, ‘being old’, having ‘an age’, and so on.

We shall now use this technique to design a “game” application. The objective of the game is to take and drop balls subject to certain rules. The following specification has been deliberately simplified.

- There are *hands* and *balls*;
- *Balls* are either used or free;
- Initially *balls* are free;
- Used *balls* cannot be taken by *hands*;
- Free *balls* can be taken by *hands*;
- If a *ball* is taken by a *hand* it becomes used;
- A *hand* can drop its *ball*;
- Dropping a *ball* makes it free;
- Each *ball* has its own *name*; and
- Each *hand* has its own *type*: left or right.

For simplicity we shall assume that any `String` may act as a valid ball name or hand type.

To find the classes in our program we need to look for the *actors* in the program. The reason why this is useful is because the actors usually correspond to the objects and each object is an instance of its class. To find the actors we look for nouns in the specification. This is called a *noun analysis*. The nouns in the previous itemised list are the italicised words. If we take the singular form of the nouns we have four nouns: ‘ball’, ‘hand’, ‘name’, and ‘type’. The nouns ‘name’ and ‘type’ are not really *active*, i.e. they don’t *do* very much. The active (singular) nouns are ball and hand. It seems reasonable to create a class for each of them. We shall call them `Ball` and `Hand`.

6.1 The `Ball` Class

Let’s first look at the `Ball` class. To see which methods and attributes are required by the `Ball` class, it makes sense to look at properties of `Balls` and for the actions of `Balls`. The following requirements seem most relevant.

- Balls are either *used* or *free*;
- Initially balls are *free*; and
- Each ball has *its own name*.

It seems like a good idea to have attributes `used` and `name` and provide getter and setter methods for these attributes. Since each `Ball` has its own name and `Balls` are initially free, it seems reasonable to have a `Ball` constructor that depends on a `String` which is the `Ball`’s name.

```

public class Ball {
    private final String name;
    private boolean used;

    public Ball( String name ) {
        this.name = name;
        used = false;
    }

    // Getter and setter methods omitted.

    @Override
    public String toString( ) {
        return "Ball[ name = " + name + " ]";
    }
}

```

For simplicity the getter and setter methods have been omitted. Notice however that the attribute `name` is a `final` attribute. This is reasonable because it is not stated that `Balls` can change their name.

6.2 The Hand Class

Next let's look at what's required for the `Hand` class. The following are the requirements that relate to properties and actions of `Hands`.

- Used balls cannot be *taken* by hands;
- Free balls can be *taken* by hands;
- If a ball is *taken* by a hand it becomes used;
- A hand can *drop its ball*;
- *Dropping* a ball makes it free; and
- Each hand has its own *type*.

The italicised words are candidate properties and actions of `Hands`. It is obvious that a hand has its own 'type' and also has 'its `Ball`', so we might just as well introduce an attribute `type` and `ball`. It is also obvious that the constructor should depend on the `Hand`'s 'type'. The things a `Hand` can do is 'take a `Ball`', 'drop its `Ball`', and 'make its `Ball` free'. The verbs are 'take' and 'drop' and each operates on a single object: a '`Ball`'. Usually it is a good idea to turn the verbs into method names and the objects into method arguments.

Let's see what we end up with if we use this as a rough implementation of our `Hand` class.


```

public class Hand {
    private final String type;
    private Ball ball;

    public Hand( String type ) {
        this.type = type;
        ball = null;
    }

    public void take( Ball ball ) { <to do> }
    public void drop( ) { <to do> }
    // Getters and setters omitted.

    @Override
    public String toString( ) {
        return "Hand[ type = " + type + ", ball = " + ball + " ]";
    }
}

```

Note that we did not implement the method `drop()` with a given `Ball` argument: `drop(Ball ball)`. This makes sense because a `Hand` always drops its current `Ball`.

We still need to complete to implement the proper rules for ‘taking’ and ‘dropping’ `Balls`. Let’s start with the method ‘`take()`’.

```

public void take( Ball ball ) {
    if (this.ball != null) {
        // We cannot take a Ball if Hand is full.
        System.err.println( "** " + this + " is full." );
        System.err.println( "** Cannot take " + ball + "." );
    } else if (ball.getUsed( )) {
        // We cannot take a used Ball.
        System.err.println( "** " + ball + " is taken." );
        System.err.println( "** Cannot take it." );
    } else {
        // Take ball.
        // Formally mark ball as used.
        ball.setUsed( true );
        // Make ball our current Ball.
        this.ball = ball;
    }
}
}

```

The implementation of the method ‘`drop()`’ turns out equally easy.

```

public void drop( ) {
    if (ball == null) {
        // We cannot drop a ball if we don't have one.
        System.err.println( "** " + this + " is empty." );
        System.err.println( "** Cannot drop any ball." );
    } else {
        // Drop our current ball.
        // Formally mark ball as free.
        ball.setUsed( false );
        // Make hand empty.
        ball = null;
    }
}
}

```

6.3 The main Method

Now that we've implemented our classes let's "play" with our Ball and Hand objects. The following is the implementation of the main.

```

public static void main( String[] args ) {
    Hand left  = new Hand( "left" );
    Hand right = new Hand( "right" );
    Ball baseBall = new Ball( "baseball" );
    Ball footBall = new Ball( "football" );

    left.take( baseBall );
    right.take( baseBall ); // Results in error message
    right.take( footBall );
    left.drop( );
    left.drop( );           // Results in error message
}

```

7 For Monday

Study the notes and Pages 71–79 from the book.