

Software Development (cs2500)

Lectures 7 and 8: Introduction to Objects (Continued)

M.R.C. van Dongen

October 11, 2010

Contents

1	Overview	1
2	Widening	2
3	Casts	2
4	String Concatenation	3
5	Flipping Bits	4
5.1	A Simple Generator	5
5.2	A Flexible Constructor	6
6	Random Values	7
6.1	Generating the Numbers	8
6.2	Seeds	9
7	Class versus Instance	9
8	Bullet Points	12
9	For Wednesday	12

1 Overview

Most of these lecture notes correspond to the last part of Chapter 2 of the book. However, some notes do not correspond to any part of Chapter 2 and, indeed, the book.

2 Widening

This section studies arithmetic expressions with mixed-type operands and automatic *widening* type conversions.

We've seen that Java is strongly typed. The type of all expressions must make "sense". Still, Java is pretty flexible. For example, you can write `'1 + 2.3'`. So how does this work?

Consider the expression `'⟨expr⟩1 + ⟨expr⟩2'`, where `⟨expr⟩1` and `⟨expr⟩2` are numeric expressions and where the type of `⟨expr⟩1` is `⟨type⟩1`, and the type of `⟨expr⟩2` is `⟨type⟩2`. Consider `⟨type⟩1` and `⟨type⟩2` and choose the type with maximum "magnitude". Let `⟨type⟩` be that type. The expression `'1 + 2.3'` is evaluated using the type `⟨type⟩`. However, this may involve converting `⟨expr⟩1` and/or `⟨expr⟩2` to `⟨type⟩` if `⟨type⟩1` and/or `⟨type⟩2` have a smaller magnitude than `⟨type⟩`. Next the resulting expressions are added. The result has the type `⟨type⟩`.

This works similar for other arithmetic operators.

For example, consider the expression `'1 + 2.3'`. The expression `'1'` is an `int` literal. The expression `'2.3'` is a `double` literal. The type `double` has the maximum magnitude. Java automatically converts the `int` to a `double`. (This conversion is called *widening*.) The result of this *widening* conversion is `1.0`. Next the operator is applied. This results in `3.3`.

Widening of primitive type expressions never loses magnitude information. However, it may lose information because of rounding. Here rounding may occur with the following conversions: `int` or `long` to `float`, and `float` to a `double`. Still it is guaranteed that the source value is converted to *some* — not *the* — nearest possible target value.

Table 1 lists the possible combinations of primitive source and target types for widening operations. The table only lists the non-trivial type combinations (where the source and target types differ).

Source Type	Target Type					
	short	char	int	long	float	double
byte	✓	✓	✓	✓	✓	✓
short			✓	✓	✓	✓
char			✓	✓	✓	✓
int				✓	✓	✓
long					✓	✓
float						✓

Table 1: Source and target types for non-trivial widening operations.

3 Casts

As you may recall from Lecture 3, some primitive types require more storage space than others.

Java does not allow assignments to primitive type variables if, based on type information, there is the possibility of loss of magnitude.

- For example, you cannot assign floating point values to integer variables.

- Likewise you cannot assign an integer value to an integer variable whose size (in bits) is smaller than the size that is required to represent the value.

The same is true for arguments of methods. For example, you cannot pass a long value as an actual parameter that corresponds to an int formal parameter.

The compiler only uses the information about the types of the values and variables that are involved. This should explain why the following is not allowed: a long cannot be assigned to an int, regardless of the value of the long.

```
long longVar = 0;
int  intVar  = longVar;
```

Don't Try this at Home

The following demonstrates how to convert the long value to an int.

```
long longVar = 0;
int  intVar  = (int)longVar;
```

Java

Basically, the expression '(int)longVar' tells the compiler: "Trust me, I know longVar is a long, but just convert its value to an int.". Explicit type conversions of the form '(<type>) <expression>' are called *casts*. They take the value of <expression> and convert this value to the type <type>.

Casting is right associative. If you think about it this makes sense:

```
int number = (int)(double)(float)1;
// same as: int number = (int)( (double)( (float)1 ) );
```

Java

Casting has the highest possible precedence. Therefore, the following won't work:

```
int number = (int)0.0 + 1.0;
```

Don't Try this at Home

Adding parentheses solves the problem.

```
int number = (int)(0.0 + 1.0);
```

Java

Numeric-to-numeric casts are always allowed. However, casts of the form primitive type-to-object are not allowed. Likewise, casting from object type to primitive type is also not allowed. For the moment we shall forget about other casts.

4 String Concatenation

Let <string>₁ and <string>₂ be two strings. The following may be used to concatenate the two strings:

'<string>₁ + <string>₂'.

The following demonstrates how this may be used in a Java program.

```
String hello = "Hello";
String world = "world";
System.out.println( hello + " " + world );
```

Java

You may also use ‘+’ in combination with a string and a numeric value. In this case, the numeric value is automatically converted to string and then the two strings are concatenated. The following is an example.

```
System.out.println( 9 + " = " + 10 + " - " + 1 );
System.out.println( 1 + 2 + " = " + 3 );
System.out.println( "" + 1 + 2 + " = " + 12 );
```

Java

To understand this example, note that ‘+’ is left associative, so the example is equivalent to the following.

```
System.out.println( (((9 + " = ") + 10) + " - ") + 1 );
System.out.println( ((1 + 2) + " = ") + 3 );
System.out.println( ((("" + 1) + 2) + " = ") + 12 );
```

Java

Notice that the ‘+’ operator can be used for different types of arguments:

- ‘ $\langle \text{numeric type} \rangle_1 + \langle \text{numeric type} \rangle_2$ ’;
- ‘ $\langle \text{numeric type} \rangle + \text{String}$ ’;
- ‘ $\text{String} + \langle \text{numeric type} \rangle$ ’; and
- ‘ $\text{String} + \text{String}$ ’.

We say that the operator is *overloaded*.

5 Flipping Bits

This section is not in the book. It explains how to implement a simple bit-sequence generator object. We shall implement the generator using a class called `BitSequencer`. The class has an instance method ‘`int nextBit()`’ which returns the next bit in the sequence. The first bit is 1. The next bit is always the “opposite” of the previous bit, so given initial bit sequence

$$1, 0, 1, \dots, b,$$

the next bit is given by $1 - b$.

5.1 A Simple Generator

In this section we shall implement the simple generator. Before we start, let's think about the *state* and *behaviour* of our `BitSequencer` objects. The *state* is what the object knows. In our case we need to know something that lets us compute the next bit in the sequence. For a newly created object the next bit is 1. For “used” objects it is $1 - b$, where b was the last returned bit. State is always implemented using *attributes* (*instance variables*). We can represent our state using a single `int` attribute. Let's call the attribute `nextBit`. We initialise it to 1. When returning the next bit we flip the attribute's value.

To tell the java compiler that `nextBit` is an attribute, we simply declare it at the top level in the class file.

```
public class BitSequencer {  
    private int nextBit;  
}
```

Java

Initialising the attributes of newly created object is done by the JVM. To tell the JVM *how* to create (construct) new objects, you provide a definition of constructor method for the object. Inside the body of the constructor method you write the statements that initialise the object's attributes.

The name of a constructor method is always the same as that of its class. Constructor methods have no return value, so they don't have a return type.

In our case our object is a `BitSequencer` so we have to provide a definition of a `BitSequencer` constructor. Our `BitSequencer` object has one attribute and we initialise it in the body of the `BitSequencer` constructor. This may be implemented as follows:

```
public class BitSequencer {  
    private int nextBit;  
  
    public BitSequencer( ) {  
        nextBit = 1;  
    }  
}
```

Java

The *behaviour* is what the object does: returning the next bit. We were told the name of the method that returns the next bit is `nextBit()`. We were also told the method returns an `int`.

Behaviour is always implemented using *instance methods*. To implement the `nextBit()` behaviour, you define an instance method called `nextBit()`. This may be done as follows.

```

public class BitSequencer {
    private int nextBit;

    public BitSequencer( ) {
        nextBit = 1;
    }

    public int nextBit( ) {
        int result = nextBit; // initialise return value.
        nextBit = 1 - nextBit; // "update" nextBit.
        return result;        // return next bit in sequence.
    }
}

```

Having defined the `BitSequencer` class, we can now use the class to create `BitSequencer` objects and use these objects to generate bit sequences.

```

public class Main {
    public static void main( String[] args ) {
        BitSequencer seq1 = new BitSequencer( );
        BitSequencer seq2 = new BitSequencer( );

        System.out.println( seq1.nextInt( ) ); // prints 1
        System.out.println( seq1.nextInt( ) ); // prints 0
        System.out.println( seq1.nextInt( ) ); // prints 1
        System.out.println( seq2.nextInt( ) ); // prints 1
    }
}

```

Note that each `BitSequencer` object has its own state: they don't share their attribute.

5.2 A Flexible Constructor

In our previous implementation of the `BitSequencer` class, we initialised the `nextInt` attribute by assigning it the value 1. As a *default* implementation this may be reasonable. However, what if a user wishes a different initial value? Clearly changing the implementation of the `BitSequencer` constructor by assigning 0 to `nextInt` won't solve the problem. (Because then it's impossible to start the sequence with 1.)

To overcome problems like this, Java lets you write several different constructors per class. In the following we add a constructor to the `BitSequencer` class. We keep our default constructor that initialises `nextBit` to 1. However, we also provide a method that initialises `nextBit` to a specific value.

```

public class BitSequencer {
    private int nextBit;

    public BitSequencer( ) {
        nextBit = 1;
    }

    public BitSequencer( int initial ) {
        nextBit = initial;
    }
}

```

For sake of the example, it is assumed that `initial` is a valid value.

6 Random Values

An infinite integer sequence

$$n_0, n_1, \dots$$

is called a *random* integer sequence if you cannot predict the value of any member of the sequence from the values before that member in the sequence. For example, it should be impossible to predict the value of n_2 using the values of n_0 and n_1 .

It should be noted that the word ‘random’ does not provide any clues about how many times a given number is included in the first m members of a random integer sequence. For example, some numbers may occur more frequently than others. However, for the purpose of this section it is assumed that if i and j are two integers in the sequence, then

$$\lim_{m \rightarrow \infty} \frac{|\{n_k : 0 \leq k < m, n_k = i\}|}{|\{n_k : 0 \leq k < m, n_k = j\}|} = 1,$$

i.e. the number of occurrences of i and the number of occurrences of j in the first m integers in the sequence is about equal as m becomes large.

It is impossible to generate random integer sequences but it is possible to generate *pseudo-random* integer sequences. Here a sequence is called *pseudo-random* if it is “difficult” to predict the members of the sequence.

Pseudo-random numbers have many applications. For example, by providing a little (pseudo-)randomness it makes it possible to implement an element of unpredictability, which may be a requirement for certain games. Randomness also makes it possible to make certain computations more robust — in the sense that they require less time on the average.

A *pseudo-random number generator* is an object that lets us generate a pseudo-random number sequence. This works just as with our `BitSequencer` object:

- There’s a constructor method for the generator (an object).

- You use the object’s instance method “next()” to get the first number in the sequence.
- Subsequent numbers are also obtained with “next()”.

Implementing a pseudo-random number generator is by no means a trivial exercise. Fortunately, Java provides good pseudo-random number generators which let you compute pseudo-random floating point numbers as well as pseudo-random integers. There are two different approaches to generating random numbers:

1. Use a dedicated object from the Random class. Generate the random numbers using the object’s (instance) methods.
2. Use the method ‘static double random()’ from the Math class. This method can be used without referring to any object: it is a class method.

6.1 Generating the Numbers

The following demonstrates how you generate pseudo-random integers using objects from the Random class. The first line of the listing is special — it is called an *import* statement. An import statement tells the java compiler that you want to use one or more non-standard classes. The import statement in the example tells the java compiler that we want to use the non-standard class called Random, which is part of the package java.util. Import statements should always occur at the start of a java source file (even if it includes JavaDoc comments).

```
import java.util.Random;
:
Random rand = new Random( );
int random1 = rand.nextInt( );
int random2 = rand.nextInt( <positive int> );
```

Java

The instance method ‘int nextInt()’ of the Random class returns a pseudo-random int value. The result may be positive, zero, or negative. The instance method ‘int nextInt(<positive int>)’ returns a pseudo-random int value in the range 0, ..., <positive int> – 1.

As already mentioned, you may also create random numbers with the *class* method ‘static double random()’ from the Math class. In its basic form the call Math.random() returns a non-negative pseudo-random double value which is less than 1.0. The following shows how to use the method to construct random non-negative integer, i, such that $0 \leq i \ \&\& \ i < 8$.

```
int i = (int)(8 * Math.random( ));
```

Java

To see how this works, notice that the call Math.random() returns a double, *d*, such that $0 \leq d \ \&\& \ d < 1.0$. After multiplying it by 8 we get a non-negative double which is less than 8. Casting this double to an int we get a random int in the range 0–7.

6.2 Seeds

It is also possible to create `Random` objects as follows:

```
'new Random( seed )'.
```

Here `seed` is a `long` value that completely determines the resulting pseudo-random number sequence.

If you create a `Random` object with `'new Random()'` then the seed which is used to create the object is not specified explicitly. Instead, you rely on an implicit seed. If you create `Random` objects with an implicit seed, the seed that is used to create the object typically depends on the time at which the object was created.

When testing, you typically want *reproducible* results. So, if you just use `'new Random()'` your results are typically *not* be reproducible. This makes testing more When creating `Random` objects, you should therefore make sure that the creation depends on the state of the software: test-mode versus production-mode.

The following shows the mechanism. We define two class methods which may be used to create `Random` objects. Next we define two *class* variables called `testing` and `defaultSeed`. In test mode, the attribute `testing` should be `true`; otherwise `false`. The class methods that create the `Random` objects use `defaultSeed` if `testing` is `true`. This makes the resulting pseudo-random number sequences predictable, which usually makes testing easier. If `testing` is `false` then the class method `myRandom()` returns a `Random` object with a seed that depends on the time and the class method `myRandom(long seed)` returns a `Random` object which depends on the seed `seed`.

```
private static boolean testing = <boolean value>;
private static long defaultSeed = <long value>;

public static Random myRandom( ) {
    return testing
        ? new Random( defaultSeed )
        : new Random( );
}

public static Random myRandom( long seed ) {
    return testing
        ? new Random( defaultSeed )
        : new Random( seed );
}
```

Java

7 Class versus Instance

This section provides a little more insight in the difference between class and instance methods and class and instance variables.

As we've seen, Java has *class* and *instance* methods. There are also *class* and *instance* variables. *Class methods* and *class variables* are owned by the class. There is one method/variable per class. *Instance*

methods and *instance variables* are owned by instances. (Remember that an object is an *instance* of its class. Hence the name instance method/variable.) There is one method/variable per instance of the class.

The notation you use for class methods depends on where “you” are. You may always write ‘<class>. <method>(<arguments>)’. However, inside the defining class you may also write ‘<method>(<arguments>)’. For variables this works similarly, so you may always write ‘<class>. <variable>’. Inside the defining class you may also write ‘<variable>’.

The following two classes should demonstrate the difference. First consider the following class.

```
public class Inside {
    public static int variable;

    public static void method( ) {
        int var1 = variable;
        int var2 = Inside.variable;
        System.out.println( var1 + " = " + var2 );
    }
}
```

Inside the class we have two notations for the class variable `variable`. Outside the class, we only have one notation.

```
public class Outside {
    public static void method( ) {
        // System.out.println( variable ); // Not allowed.
        System.out.println( Inside.variable );
    }
}
```

Notice that the ‘`System.out`’ also demonstrates the notation because `out` is a class variable of the class `System`.

The dot-notation for instance variables and methods is similar. You may always use ‘<reference>. <method>(<arguments>)’. Here <reference> is an object reference — usually an object reference variable. However, inside the defining class you may also write ‘<method>(<arguments>)’. For attributes this works similarly, so you may always write ‘<reference>. <variable>’. But inside the defining class you may also write ‘<variable>’.

The dotless notation for instance variables and instance methods is only allowed inside instance methods. Inside a given instance method you may use the notation ‘`this`’ for the “current” object, i.e. the object that called the method: it owns the method. Using ‘<instance variable>’ without dot-notation is shorthand notation ‘`this.<instance variable>`’. For instance methods this is the same. So ‘<instance method>(<arguments>)’ means ‘`this.<instance method>(<arguments>)`’.

The following example should explain the difference. The two instance methods are (effectively) identical.

```

public class Inside {
    private int attribute;

    private static void classMethod( int var ) {
        System.out.println( var );
    }

    public void instanceMethod1( ) {
        classMethod( attribute );
    }

    public void instanceMethod2( ) {
        classMethod( this.attribute );
    }
}

```

```

public class Outside {
    public static void main( String args[] ) {
        Inside inside = new Inside( );
        inside.instanceMethod1( );
        inside.instanceMethod2( );
    }
}

```

You can always simulate instance methods with class methods. However, this comes at the price of an extra parameter to represent the “current” object (*this*). The following two classes should make this clear.

```

public class Simulation {
    private int attribute;

    public static void classMethod( Simulation current ) {
        System.out.println( current.attribute );
    }

    public void instanceMethod( ) {
        classMethod( this );
    }
}

```

```

public class Main {
    public static void main( String args[] ) {
        Simulation simulation = new Simulation( );
        // The following calls are effectively identical.
        simulation.instanceMethod( );
        Simulation.classMethod( simulation );
    }
}

```

Java

To see how this works, notice that the instance method `instanceMethod` is called in the `main`. It is called using the dot-notation, so the method is called by “remote control”. The object that calls the method is the object which is referred to by the variable `simulation`. This makes that object the “current” object in the instance method. So effectively, “`this`” in the instance method and `simulation` in the `main` refer to the same object. Next, the instance method in the `Simulation` class calls the class method with `this`. This is equivalent to calling the class method with ‘`simulation`’ as its argument, which is the second call in the `main`.

8 Bullet Points

The following are some of the more important concepts from Chapter 2.

- OOP lets you extend a program without having to touch previously tested, working code.
- All Java code is defined in a class.
- A class is an *object blueprint*.
- Objects govern themselves. They take care of themselves. How they do this shouldn’t matter to you.
- Objects have *state* and *behaviour*.
- The state is determined by the *instance variables*.
- The behaviour is determined by the *methods*.
- A class may *inherit* instance variables and methods from a more abstract *superclass*.

9 For Wednesday

Study Chapter 2, solve the puzzles from Chapter 2, and carry out the exercises from Chapter 2.