

Ian Kenny

October 3, 2017

# Software Workshop

## Lecture 2b

# In this lecture

- Coding conventions: Layout, Naming, Magic numbers, Constants, Comments.
- Validation.
- Basic testing introduction.

# Code conventions

When writing software it is very important that we remember that, in general, other developers will need to read, understand and amend our code. It is rare in the corporate world for a developer to work alone on a program. You will need to understand this and develop a style that reflects it.

Not only that, when working in a team of developers, certain conventions must be adopted that are then followed by all of the developers. These conventions can cover a very wide range of aspects of the development task. These conventions are adopted so that developers share certain expectations about the *codebase*. For example, that variable names will follow a particular pattern. These conventions add to the *shared language* aspect of team development. They are not part of the programming language hence can only be adopted by convention.

# Code conventions

In the case of variable names, for example, the Java compiler only requires you to follow its rules (we will look at those shortly), but that can still lead to hard to understand naming. Naming conventions can help with the readability of the code.

One of the most important but perhaps obvious points about conventions is that once a convention has been selected you should stick to it. There are various 'competing' conventions for different aspects of the programming task. More important maybe than *which* one is chosen is that once it is chosen it is used consistently.

The overall goal is *readability*. This may seem an ill-defined term but there are some steps we can take that undeniably improve it.

# Layout

Firstly we will look at *layout* which is about how different parts of the program are positioned in the source code, where braces are placed, how much white space should be used and where, etc. Good layout significantly increases the readability of code. Conversely, poorly laid out code can make even the simplest programs hard to understand.

# Hello, Confusing World: demo

---

```
public class HelloConfusingWorld{public static void  
    main(String [] args){System.out. println (" Hello, Confusing  
    World.");}}
```


---

Listing 1 : HelloConfusingWorld.java

The Java compiler doesn't mind if your entire program is on one line or however else it is laid out as long as it is *syntactically correct*.<sup>1</sup>

The insertion of white space, and the use of a nice layout, are simply for us humans.

---

<sup>1</sup>In the editor, the program above is indeed written all on one line. 

# Confusing Layout: demo

What does this program print out?

---

```
public class ConfusingLayout {
public static void main(String[] args){

    int a = 7;
        int b = -7;

                                if      (a < b) {
                                    System.out.println("huh?")
                                ;
                                }
else if (a == -b)
{
a = -b;
    } else if (b < 10)
    {
        b = 12;
        System.out.println("I'm lost now");}
    }
}
```

---

Listing 2 : ConfusingLayout.java



# Layout: conventions

Firstly we will consider *block* style. Block style dictates the layout of the major structural components of programs. In the case of Java, these are delineated by braces { }. Thus, classes, methods, if statements, while loops, etc. are all delineated by braces.

One of the most popular style conventions is the so-called K&R style (Kernighan and Ritchie), although many teams use a variant of that style which is illustrated on the next slide.

# Block layout: demo

---

```
public class LayoutTest {  
    public static void main(String[] args) {  
        int num1 = 0;  
        int num2 = 0;  
  
        num1++;  
  
        int num3 = (num2 + 2) * num1;  
  
        if (num3 > 2) {  
            System.out.println("yes");  
        } else {  
            System.out.println("no");  
        }  
    }  
}
```

---

Listing 3 : LayoutDemo1.java

# Brace layout

On the previous slide, concentrate on the braces. The opening brace of each block is on the same line as the 'header' of the block it is opening (in this example, the class, the `main` method and the `if-else` block). The closing brace, however, is in the same *column* as the start of the 'header' but on a *new line*.

You **must** adopt the layout convention shown above.

# Tabs

The other major layout element demonstrated in the previous listing is the use of *tabs*. Pressing the tab key inserts a certain number of spaces into a line.

Each time a block is opened, the next line of code, and all others in the same block, should be indented by *one tab*. This is recursive, so if a new block is opened inside *that* block, the next line and everything in the new block are also tabbed in, and so on. Have another look at the previous listing to see this in action.

You **must** adopt the convention that a tab consists of **four spaces**. This is configurable in most editors. Do not use an editor that doesn't let you configure this (unless it defaults to four spaces).

# Naming

Another major contributor to the readability of source code is in the naming of entities in the code. Entities include classes, methods, variables, constants, etc. That is, the things that *can* be named.

Consider the next listing.

# Naming: demo

What does this command line program do?

---

```
public class NamingDemo1 {  
    public static void main(String[] args) {  
  
        String a = args[0];  
        int b = Integer.parseInt(a);  
        int c = b/14;  
        int d = b%14;  
        System.out.println(c + " " + d);  
    }  
}
```

---

Listing 4 : NamingDemo1.java

Note that `Integer.parseInt()` converts a `String` to an `int`.

# Naming

The program takes a weight in pounds and outputs the same weight in stones and pounds. It isn't very clear from the program that this is what it does, partly due to the naming of the variables. The names `a`, `b`, etc. are not very descriptive, to say the least.

Consider the amended version of that program on the next slide.

# Naming: demo

---

```
public class NamingDemo2 {  
    public static void main(String[] args) {  
  
        String inputWeight = args[0];  
        int totalPounds = Integer.parseInt(inputWeight);  
        int stones = totalPounds / 14;  
        int pounds = totalPounds % 14;  
        System.out.println(stones + " " + pounds);  
    }  
}
```

---

Listing 5 : NamingDemo2.java

The new names describe the operation of the program far better.



# Naming: rules (and conventions)

Which names can and should you use?

Java does have its own rules on characters you may use in names and names that are simply not allowed.

The names that aren't allowed are all of the Java *reserved words*. These include words like `class`, `float`, ...

Characters that may be used are the alphabetic and numeric characters, and the underscore character. Officially, the dollar sign may also be used but conventionally it is never used.

Names may not start with a numeric character and conventionally don't start with an underscore.

All of Java is case sensitive including reserved words and names.

# Naming: mostly bad (part one)

---

```
public class NamingDemo5 {  
    public static void main(String[] args) {  
  
        int $ = 6; // legal. Don't do it.  
        int $$$ = 1000000; // legal. Don't do it.  
        int $perfectlyValidName = 3; // don't do it.  
  
        int y = -4; // what is this for?  
        int yCoordinate = -4; // much better.  
  
        int class = 9; // NO!  
        int public = 34; // NO!
```

---

Listing 6 : NamingDemo5.java

# Naming: mostly bad (part two)

---

```
int validbutnotveryeasytoread = 0; // don't do it.  
float former_c_language_programmer = 2; // I don't  
    like this.
```

```
String _____ = "HUH?";  
String _____ = "I HAVE NO IDEA.";  
String _____ = "NOR ME.";
```

```
int 9miles = 9; // NO!
```

```
float StOPThaTRighTNOw = 3.14; // Don't do it.
```

```
String you cannot have spaces in names = "FORGET  
    IT";
```

```
}
```

```
}
```

---

Listing 7 : NamingDemo5.java

# Naming and 'Camel Case'

The question arises: what case should you use for names?

Let's deal with *non-class* names first. If a name is a single word, it should have *all lower case characters*.

If a name contains multiple words, it should follow the *lowerCamelCase* convention: the initial letter of the first word should be lower case but the initial letter of all subsequent words should be upper case.

Now for classes. Single word class names should start with an upper case character.

Multiple word class names, however, follow the *UpperCamelCase* convention. This is the same as *lowerCamelCase* except that the first character of the first word also starts with an upper case character.

# Naming

Names should not be so short (in most cases) that the reader cannot discern what the point of them is.

Conversely, they should not be too lengthy either.

The art is in choosing the shortest possible name you can that will be *meaningful* to other developers reading your code.

Spelling mistakes in names are really bad. Check your spelling.

# Naming

---

```
public class NamingDemo6 {  
    public static void main(String[] args) {  
  
        int a = 0; // probably too short.  
  
        int theMaximumValueOfX = 0; // probably too long.  
  
        int xMax = 0; //better.  
  
        float iWillPutTheAnswerInThisVariable = 0; // No, obv.  
  
        float result = 0; // better.  
  
        String whatToSayIfSomethingWentWrong = "ERROR"; // bad.  
  
        String errorMessage = "ERROR"; // better.  
  
        int produt = 0; // what?  
    }  
}
```

---

Listing 8 : NamingDemo6.java

# Naming: what does this program do?

---

```
public class NamingDemo4 {  
    public static void main(String[] args) {  
  
        String a = args[0];  
        int b = Integer.parseInt(a);  
        if (b % 4 != 0) {  
            System.out.println("no");  
        } else if (b % 400 == 0) {  
            System.out.println("yes");  
        } else if (b % 100 == 0) {  
            System.out.println("no");  
        } else {  
            System.out.println("yes");  
        }  
    }  
}
```

---

Listing 9 : NamingDemo4.java

# Naming: what does this program do?

It outputs "yes" if the argument is a leap year and "no" if it isn't.

It can be improved with better naming.



# Naming: what does this program do?

---

```
public class IsLeapYear {
    public static void main(String[] args) {

        String inputYear = args[0];
        int year = Integer.parseInt(inputYear);
        if (year % 4 != 0) {
            System.out.println("no");
        } else if (year % 400 == 0) {
            System.out.println("yes");
        } else if (year % 100 == 0) {
            System.out.println("no");
        } else {
            System.out.println("yes");
        }
    }
}
```

---

Listing 10 : IsLeapYear.java

# Naming: what does this program do?

---

```
$ javac IsLeapYear.java
$ java IsLeapYear 2016
yes
```

---

## Listing 11 : Terminal commands

This is an improvement, and the only changes made were to names. The names in this program *document* its purpose and use better.

# Naming: demo

Look again at this program. What does the number 14 mean in this context?

---

```
public class NamingDemo2 {  
    public static void main(String[] args) {  
  
        String inputWeight = args[0];  
        int totalPounds = Integer.parseInt(inputWeight);  
        int stones = totalPounds / 14;  
        int pounds = totalPounds % 14;  
        System.out.println(stones + " " + pounds);  
    }  
}
```

---

Listing 12 : NamingDemo2.java

# Magic numbers

Once we know what the program does, we can deduce that 14 is the number of pounds in a stone but it is not obvious.

The constant 14 in this case is an example of a *magic number*. In the programming field, a magic number is a constant in a program that does not have a name and hence could almost mean anything. At the very least, we may be left scratching our heads for a while to think what it might mean.

Magic numbers should be avoided and can be avoided by declaring *constants*

# Naming: demo

---

```
public class NamingDemo3 {  
    public static void main(String[] args) {  
  
        final int POUNDS_PER_STONE = 14;  
  
        String inputWeight = args[0];  
        int totalPounds = Integer.parseInt(inputWeight);  
        int stones = totalPounds / POUNDS_PER_STONE;  
        int pounds = totalPounds % POUNDS_PER_STONE;  
        System.out.println(stones + " " + pounds);  
    }  
}
```

---

Listing 13 : NamingDemo3.java

In Java, constants are declared with the word `final`. Note the all upper case letters in the name and the underscores. This is again a convention and is, in fact, a hangover from C language pre-processor directive `#define` which could be used to define constants.

# Comments

Comments are text in the source code that is not part of the program. The ability to put comments in code is useful because you can use them to explain how aspects of the code work, explain otherwise obtuse parts of the code, draw attention to particular, noteworthy cases, etc.

Comments should not be overused though. Not every line of code needs a comment, particularly if names are good and the code is clearly written.

# Comments

Comments can be single line or multi-line.

---

```
public class CommentsDemo0 {  
    public static void main(String[] args) {  
  
        // this is a single line comment  
  
        /* this is a  
           multi-line  
           comment  
        */  
  
    }  
}
```

---

Listing 14 : CommentsDemo0.java

Consider the program on the next slide. Do all of these comments seem necessary?

# Comments: demo (part one)

---

```
public class CommentsDemo1 {  
    public static void main(String[] args) {  
  
        // declare an integer and set it to 5  
        int numIterations = 5;  
  
        /* declare a String that will hold an ID number  
           and set it to empty */  
        String idNumber = "";  
  
        /* declare an integer to use as an index and call  
           it index */  
        int index;  
  
        // initialise the index variable to zero.  
        index = 0;  
    }  
}
```

---

Listing 15 : CommentsDemo1.java



# Comments: demo (part two)

---

```
// start a while loop that will loop 5 times
while (index < numIterations) {

    //create a new ID new number
    String newIDNumber = idNumber + index;

    // set the base ID number
    idNumber += index;

    //print out the new ID number
    System.out.println(newIDNumber);

    // increment index to the next value with ++
    index++;
}
}
```

---

Listing 16 : CommentsDemo1.java

# Comments

The problem with the comments in this program is that they

- Explain very basic statements that anyone reading the source code should understand.
- Are too verbose.
- They distract attention from the code itself.

Comments should only provide information that a programmer reading the code may not be aware of, may find helpful, etc.

Sometimes the comments will be aimed at the person who wrote them, i.e. as reminders about why something was done or how it works. At other times (the usual case) the comments will be aimed at others reading the code. The comments should always simply aim to be helpful and should never be used *in place of* good naming, structure, style, etc.

Consider the refactored version of the previous program, on the next slide. Only information that is not inherent in the program is given.

# Comments: demo

---

```
public class CommentsDemo2 {  
    public static void main(String[] args) {  
  
        // only need 5 ID numbers  
        int numIterations = 5;  
  
        String idNumber = "";  
  
        int index = 0;  
  
        while (index < numIterations) {  
  
            // ID number format = previous ID + previous ID + index  
            String newIDNumber = idNumber + index;  
  
            idNumber += index;  
  
            System.out.println(newIDNumber);  
  
            index++;  
        }  
    }  
}
```

---

Listing 17 : CommentsDemo2.java

# Comments: see anything wrong?

---

```
public class CommentsDemo2 {  
    public static void main(String[] args) {  
  
        // only need 5 ID numbers  
        int numIterations = 6;  
  
        String idNumber = "";  
  
        int index = 0;  
  
        while (index < numIterations) {  
  
            // ID number format = previous ID + previous ID + index  
            String newIDNumber = idNumber + index;  
  
            idNumber += index;  
  
            System.out.println(newIDNumber);  
  
            index++;  
        }  
    }  
}
```

---

Listing 18 : CommentsDemo3.java

# Javadocs

Javadocs is an application that allows us to create professional looking code documentation from comments in our code. It is best applied to classes, however, so we will need to revisit Javadocs when we have covered classes.

# Summary of conventions

You must follow the conventions in the previous slides. Here is a summary:

- You must use the *layout* demonstrated above.
- You must use *tabs* correctly and they must consist of four spaces.
- Names must be sufficiently descriptive but not too verbose.
- Do not use underscore (except for constants. See below) and the dollar sign in names.
- Names should not contain spelling mistakes.
- All names must follow the appropriate *camel case*.
- All constants should be properly defined as `final` (i.e. no magic numbers).
- Constant names should be all upper case. You can use underscores in constant names.
- Comments must be judiciously used (i.e. not overused or underused).

# Validation

If any part of a program can receive data, it might be the case that the data needs to be *validated*.

Validation means checking that the data has a valid value. The definition of 'valid' depends on the application. Valid values are often expressed in a range. Valid values lie inside the range, and invalid values lie outside the range.

For example, what might be a valid range for an 'age' value?

Would it **always** be the case that negative ages are invalid? What about ages over 18? Ages over 100?

# Validation

You, the programmer, must generally leave behind your own conceptions of validity.

The valid range of some data will be specified by whoever has designed the application.

If no valid range is specified for some data item then, yes, you may exercise your judgement. But, your assumptions should be stated and justified, and the justification should have a rational basis.



# Validation: what do you think of this (1)?

---

```
import java.util.Scanner;

public class ValidationExample {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter your name.");
        String inputLine = in.nextLine();

        if (inputLine.length() > 7) {
            System.out.println("Error: your name is too long.");
        }

        else System.out.println("Hello, " + inputLine);
    }
}
```

---

Listing 19 : ValidationExample.java

# Validation: what do you think of this (2)?

---

```
import java.util.Scanner;

public class ValidationExample2 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter your name.");
        String inputLine = in.nextLine();

        if (inputLine.charAt(0) == 'Z') {
            System.out.println("Error: no names begin
                               with the letter Z.");
        }
        else System.out.println("Hello, " + inputLine);
    }
}
```

---

Listing 20 : ValidationExample2.java

# Validation: what do you think of this (3)?

---

```
import java.util.Scanner;

public class ValidationExample3 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter your name.");
        String name = in.nextLine();
        System.out.println("Enter your age.");
        int age = in.nextInt();

        if (name.equals("Ian") && age < 50) {
            System.out.println("Error: nobody aged under
                               50 is called Ian.");
        }
        else System.out.println("Hello, " + name + " you
                                are " + age + " years old.");
    }
}
```

---

Listing 21 : ValidationExample3.java

# Validation: what do you think of this (4)?

---

```
import java.util.Scanner;

public class ValidationExample4 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter your age.");
        int age = in.nextInt();

        if (age < 25) {
            System.out.println("Error: you are too
                               young.");
        }

        else if (age > 25) {
            System.out.println("Error: you are too old.");
        }

        else System.out.println("WELCOME!!!");
    }
}
```

---

Listing 22 : ValidationExample4.java

# Validation

The validity of data values depends on the application.

The range of valid values will be stated in the application or should be 'obvious' from the application. If neither of those cases applies, the programmer themselves must select valid ranges (maybe in conjunction with others).

Assumptions about valid ranges should be stated. They can be stated in documentation (if available) and comments in the code.

The 'over-policing' of data values is not necessary, unless the data being absolutely valid is critical to the operation of the software system. If the system can be relaxed about a particular data item, because it will not cause problems, then perhaps it should be relaxed. This needs careful consideration though.

# Response to bad data

How should a system respond to bad data?

We will consider this in more detail when we look at *Exceptions*. but we can say something about this now. We will proceed for now as if we are not using Exceptions.

If a data item is fatal to the correct operation of a system, should the system quit? Should it report the error and seek to obtain the data again? Should it ignore it? Should it try and correct the data itself (e.g. a date)?

It may not be possible to report the error (in a way that someone who can correct it will see anyway). What should the system do in this case?

# Response to bad data

If a system has a human user, the developer should consider them when it comes to errors.

One consideration could be: what will be the least annoying way of responding to an error?

Crashes are annoying. Pop-up dialog boxes are annoying. Obtuse error messages are annoying. Making the user feel stupid is annoying.

In general, a system should permit the smoothest, least obtrusive use possible whilst still maintaining correct input data.

This is not an exact science.

# Response to bad data

Look again at the 'Validation' programs from a few slides back. What do you think of the response of the system in those cases? How could they be improved (presuming the system is actually correct)?



# What validation should you do?

When thinking about validation of data in your programs you must firstly do the forms of validation that you have been required to do. For example, for an application that process telephone numbers, you may have the requirement that all numbers start with the relevant country dialling code.

For requirements that aren't stated, you should use your judgement. Should you really enforce a particular validation check? What will happen if you don't? Those are the questions to ask.

# Testing

Programs should be thoroughly tested to ensure that they meet the requirements, i.e. the specification of what the program was designed to do, and that they are error free.

A program will be designed to have certain functionality. It should have that functionality and only that functionality. It should work in exactly the way that has been requested.

How would you know if the program does indeed meet the requirements? You test it, that's how.

# Testing

Testing can involve human testers using a system and reporting about whether it has the necessary functionality, whether it has errors, etc.

Testing can also be automated and done by other computer programs.

Either way, the testing of a program is done by someone or something else. Not by the program itself.

It is not usually a good idea to rely on the programmer of a piece of software to do all the testing of it either. They may not test it very thoroughly ...

# Testing: JUnit

We can automate testing by using libraries such as JUnit.

We will not require the use of that but you may use it yourself if you decide. Using JUnit will generally be a better solution.

JUnit doesn't really work too well for the type of programs we have seen so far (in lectures and in the exercises). It is better used for programs with multiple classes.

(If you have written your exercises in an object-oriented fashion, using methods, JUnit could be used.)

# Testing

Most of the programs so far (including the ones you have written) will be best tested with 'manual' methods, i.e. using the program with a range of inputs to see if it works.

For example, for the password question, passwords must be of a particular length. You should test what happens if you submit a password that is not of that length. The program should refuse to accept the password. It needs to have this functionality. How will you test that?