

About the required reading

All the reading described as required **is examinable**.

For **example**, you should be able to:

- ▶ briefly explain concepts like “process requirement”, “stakeholder”, “tacit knowledge”, “conceptual modelling”, “requirements allocation”, “traceability”
- ▶ constructively criticise a written requirement
- ▶ suggest several ways to elicit requirements from stakeholders

Use common sense: I will not ask “which parts of SWEBOK04 come from [Dav93]” or “in which Knowledge Area is ... discussed”.

Try to notice when different sources use terms somewhat differently: main example here is “system requirement”. Exam questions will disambiguate as necessary!

Construction: High quality code for ‘programming in the large’

Nigel Goddard

School of Informatics
University of Edinburgh

What is high quality code?

High quality code does what it is supposed to do...

... and will not have to be thrown away when that changes.

Obviously intimately connected with requirement engineering and design: but today let’s concentrate on the code itself.

What has this to do with programming in the large?

Why is the quality of code more important on a large project than on a small one?

Fundamentally because other people will have to **read** and **modify** your code – even you in a year’s time count as “other people”!

E.g.,

- ▶ because of staff movement
- ▶ for code reviews
- ▶ for debugging following testing
- ▶ for maintenance

(Exercise. Dig out your early Java exercises from Inf1. Criticise your code. Rewrite it better *without changing functionality*.)

How to write good code

- ▶ Follow your organisation's coding standards - placement of curly brackets, indenting, variable and method naming...

Coding standard example

- ▶ Suppose you are used to...

```
public Double getVolumeAsMicrolitres() {
    if (m_volumeType.equals(VolumeType.Millilitres))
        return m_volume * 1000;
    return m_volume;
}
```

- ▶ ... and you see

```
public Double getVolumeAsMicrolitres()
{
    if(m_volumeType.equals( VolumeType.Millilitres))
    {
        return m_volume*1000;
    }
    return m_volume;
}
```

- ▶ Even worse: mixed styles in one file - inevitable without standards!

How to write good code

- ▶ Follow your organisation's coding standards - placement of curly brackets, indenting, variable and method naming...
- ▶ Use meaningful names (for variables, methods, classes...) If they become out of date, change them.
- ▶ Avoid cryptic comments. Try to make your code so clear that it doesn't need comments.
- ▶ Balance structural complexity against code duplication: don't write the same two lines 5 times (why not?) when an easy refactoring would let you write them once, but equally, don't tie the code in knots to avoid writing something twice.
- ▶ Be clever, but not too clever (recursion!). Remember the next person to modify the code may be less clever than you! Don't use deprecated, obscure or unstable language features unless absolutely necessary.
- ▶ Remove dead code, unneeded package includes, etc.

Clicker: Which of these fragments is better?

1.

```
for(double counterY = -8; y < 8; counterY+=0.5){
    x = counterX;
    y = counterY;
    r = 0.33 - Math.sqrt(x*x + y*y)/33;
    r += sinAnim/8;
    g.fillCircle( x, y, r );
}
```
2.

```
for(double counterY = -8; y < 8; counterY+=0.5){
    x = counterX;
    y = counterY;
    r = 0.33 - Math.sqrt(x*x + y*y)/33;
    r += sinAnim/8;
    g.fillCircle( x, y, r );
}
```
3. They are both fine.

Clicker: Which of these fragments is better?

1. `c.add(o);`
2. `customer.add(order);`
3. They are both fine.

What else is wrong with this?

```
r = 0.33 - Math.sqrt(x*x + y*y)/33;  
r += sinAnim/8;  
g.fillCircle( x, y, r );
```

Use comments when they're useful

```
if (moveShapeMap!=null) {  
    // Need to find the current position. All shapes have  
    // the same source position.  
    Position pos = ((Move)moveShapeSet.toArray()[0]).getSource();  
    Hashtable legalMovesToShape = (Hashtable)moveShapeMap.get(pos);  
    return (Move)legalMovesToShape.get(moveShapeSet);  
}
```

and not when they're not

```
// if the move shape map is null  
if (moveShapeMap!=null) {
```

Too many comments is actually a more common serious problem than too few.

Good code in a modern high-level language shouldn't need many *explanatory* comments, and they can cause problems.

"If the code and the comments disagree, both are probably wrong"
(Anon)

But there's another use for comments...

Javadoc

Any software project requires documenting the code – *by which we mean specifying it, not explaining it.*

Documentation held separately from code tends not to get updated.

So use comments as the mechanism for documentation – even if the reader won't need to look at the code itself.

Javadoc is a tool from Sun. Programmer writes doc comments in particular form, and Javadoc produces pretty-printed hyperlinked documentation. E.g. Java API documentation at java.sun.com.

See web page for **Required reading** tutorial.

Javadoc example (from Sun)

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Rendered Javadoc (Eclipse)

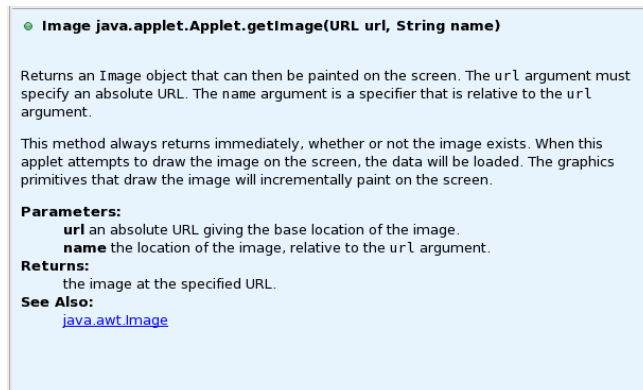


Image java.awt.Applet.getImage(URL url, String name)

Returns an Image object that can then be painted on the screen. The url argument must specify an absolute URL. The name argument is a specifier that is relative to the url argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:
url an absolute URL giving the base location of the image.
name the location of the image, relative to the url argument.

Returns:
the image at the specified URL.

See Also:
[java.awt.Image](#)

The relevance of object orientation

Construction is intimately connected to design: it is design considerations that motivate using an OO language.

Key need: control complexity and **abstract** away from detail. This is essential for systems which are large (in any sense).

Objects; classes; inheritance; packages all allow us to think about a lot of data at once, without caring about details.

Interfaces help us to focus on behaviour.

Revise classes, interfaces and inheritance in Java.

A common pattern in Java

```
interface Foo {
    ...
}

class FooImpl implements Foo {
    ...
}
```

Exercise: why is this so much used?

Example: single method interface

```
JButton button = new JButton("Example Button");

button.addActionListener(
    new ActionListener() {
        actionPerformed(ActionEvent ae) {

            // Handle the action
        }
    }
);
```

Example: GUI I

```
public class ListenExample extends JFrame implements WindowListene

    // Example WindowListener method
    public void windowIconified(WindowEvent e) {
        // Handle window minimised events
    }
    // ... Other WindowListener methods
}
```

Packages

Recall that Java has [packages](#). Why, and how do you decide what to put in which package?

Packages:

- ▶ are units of [encapsulation](#). By default, attributes and methods are visible to code in the same package.
- ▶ give a way to organize the [namespace](#).
- ▶ allow related pieces of code to be grouped together.

So they are most useful when several people must work on the same product.

But

- ▶ the package “hierarchy” is not really a hierarchy as far as access restrictions are concerned – the relationship between a package and its sub/superpackages is just like any other package-package relationship.

Packages extended example

A class `eduni.inf.jcb.MahJong` inherits from a class `eduni.inf.CardGame`. It inherits a **protected** method `deal()`. Somewhere in the `MahJong` class occurs the following code:

```
eduni.inf.CardGame c = new eduni.inf.CardGame();
c.deal();
```

Will this compile?

Does this compile?

- A Yes, and it would even if we removed **protected**.
- B Yes, but it wouldn't if we removed **protected**.
- C No, `deal()` needs to be public because `CardGame` and `MahJong` are in different packages.
- D No, and wouldn't unless we made some other change.

.....

Concretely:

In file `... eduni/inf/CardGame.java`:

```
package eduni.inf;
public class CardGame {
    protected void deal() {}
}
```

In file `eduni/inf/jcb/MahJong.java`:

```
package eduni.inf.jcb;
public class MahJong extends eduni.inf.CardGame {
    public void foo() {
        eduni.inf.CardGame c = new eduni.inf.CardGame();
        c.deal();
    }
}
```

And this?

In file `... eduni/inf/CardGame.java`:

```
package eduni.inf;
public class CardGame {
    protected void deal() {}
}
```

In file `eduni/inf/jcb/MahJong.java`:

```
package eduni.inf.jcb;
public class MahJong extends eduni.inf.CardGame {
    public void foo() {
        this.deal();
    }
}
```

.....

And this?

In file ... eduni/inf/CardGame.java:

```
package eduni.inf;
public class CardGame {
    protected void deal() {}
}
```

In file eduni/inf/jcb/MahJong.java:

```
package eduni.inf.jcb;
public class MahJong extends eduni.inf.CardGame {
    public void foo() {
        MahJong c = new MahJong();
        c.deal();
    }
}
```

.....

Suggested lab exercises

1. Revise Inf2B first couple of labs, and remind yourself how to use Eclipse to start a project and import code.
2. Play with the Umllet tool for drawing use case diagrams. Information (and downloads for home use) at <http://www.umllet.com>. Key points: to create a new diagram, use menu "File/New/Other.." then select "Umllet/Diagram". Double click on a diagram element to insert it. You have a choice of "palette"s.
3. Experiment with Java packages (see web).
4. Do the JavaDoc tutorial.

And this?

In file CardGame.java:

```
public class CardGame {
    protected void deal() {}
}
```

In file MahJong.java:

```
public class MahJong extends CardGame {
    public void foo() {
        CardGame c = new CardGame();
        c.deal();
    }
}
```

.....

Reading

Required: something that makes you confident you completely understand Java packages (e.g., see course web page).

Required: the JavaDoc tutorial

Suggested: Stevens Ch14 - 4 pages discussing UML's notion of packages. **Exercise:** compare it with Java's.