Starting Java

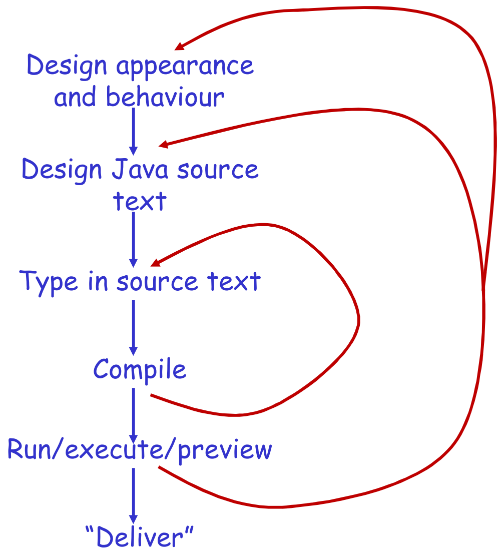
- In this section:
  - The program construction process
  - Designing a first simple application ("Greeting")
  - How the editor/files/compiler/JVM work together
  - Review and analysis of the Java code
  - A second way to build a simple application ("Hello")
- Practical use of BlueJ: In the lab sessions

The Program Construction Process

~~Simplified/idealized:~~

In practice:

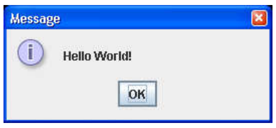
(even experienced programmers!)

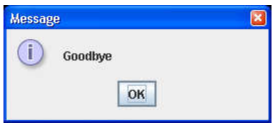


First simple application (JFS, p11)

Demo
Hello

- First we design its appearance and behaviour:





  - The left window pops up when the program starts
  - The left window disappears and the right appears (in the same place) when OK is clicked
  - The right window disappears and the program ends when OK is clicked
- Now we need to see the Java code that gives this effect when it is compiled and run on the Java Virtual machine...

The Java source text (JFS, p11)

- Without worrying about the details for the moment, the Java source code that we need is the following *text*:

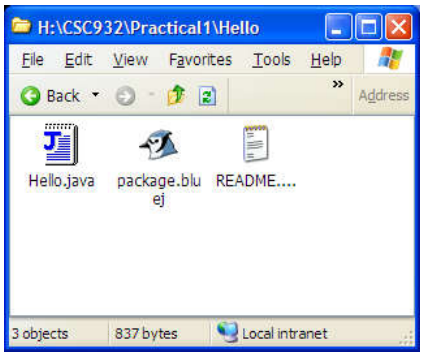
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Hello extends JFrame {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null,
                                      "Hello World!");
        JOptionPane.showMessageDialog(null,
                                      "Goodbye");
        System.exit(0);
    }
}
```

- Note: We have to type this in, or alter some existing text to look like it, *very carefully*

How the Editor/Files/Compiler/JVM inter-work

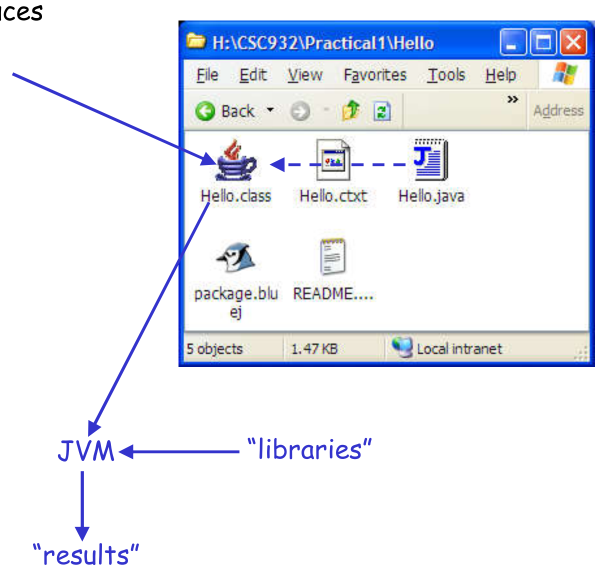
- The Java source text containing "`public class Hello`" *must* be in a file called `Hello.java`
  - We use a *text editor* to create this
  - E.g Notepad, *but BlueJ contains its own*
- The Java compiler generates the *bytecode*, and places it in a file called `Hello.class`
- The JVM finds the file `Hello.class`
  - Then carries out the instruction that it contains
  - As many times as we wish, no need to compile again!
- How it all looks is shown on the following slides…

- Freshly entered project:



- Note:
  - In its own folder
  - The *icons* may well not be anything special:
    Need to see the *file name extensions* `.bluej`, `.java`, etc (see Tools/Folder options)
  - `package.bluej` is a BlueJ project settings file - does not itself contain any Java – not interesting!
    There may be others too: `.ctxt`…

- Compiling produces a new file **Hello.class** (or overwrites an old one)



- Running:

Analysing the Java source code

- Annotating with line numbers *for reference only:*

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  public class Hello extends JFrame {
6.
7.      public static void main(String[] args) {
8.          JOptionPane.showMessageDialog(null,
                                          "Hello World!");
9.          JOptionPane.showMessageDialog(null,
                                          "Goodbye");
10.         System.exit(0);
11.     }
12. }
```

- The *layout* is "free-format" ... although the text above is arranged following a *conventional style* (more about this later)

There are Always Errors!

- The rules of Java's syntax (*grammar*) are really strict
- Common typing errors include:
  - `,` or `.` or `;` missing, extra or in wrong place
  - `{ }`s, `( )`s or `[ ]`s missing, extra or in wrong place
  - `'` instead of `"`
  - Incorrect upper/lower case
- Pay great attention to this kind of detail when typing
- The compiler will try to indicate where syntax errors are
  - But often it cannot tell where the real mistake is
  - A common problem is a missing `;` at the end of a line, but usually the compiler will report this as an error in the following line!
- You may need to repeatedly edit/compile until all syntax errors are gone - this is quite normal, and you get better!

Key points

- Lines 8 and 9: `JOptionPane.showMessageDialog(…);`
  - Only these lines actually *do* anything!
  - Is it roughly clear, from the design, what they do??!
- The rest is necessary formality
  - Relevant and more meaningful in larger applications
  - Java is a full strength industrial programming language
  - We will use standard frameworks – taken on trust for now!
- Lines 1 - 3: `import...`
  - These indicate which standard Java *libraries* are required (**`awt`** = basic Abstract Window Toolkit, **`swing`** = libraries containing elegant/advanced window components)
- Line 5: `public class Hello...`
  - Announces (and *names*) the program application,
  - which runs from the `{` on line 5 to the `}` on line 12

- Line 7: `public static void main...`
  - Announces the section of the program specifying the main action to be carried out by the JVM
  - which runs from the `{` on line 7 to the `}` on line 11
- Line 10: `System.exit(0);`
  - Causes the program to stop running
  - (Aside: 0 is a "result code" with conventional meaning "successful")
- Lines 8, 9, 10 form a *sequence: carried out strictly in order*
  - Sensible?
  - Note: each ends with a semicolon `;`
- Many parts of this program are *fixed* and we have no choice about them
  - We can choose the program name (`Hello`), and the quoted messages
  - We can add more actions to the main sequence

"Objects" and action requests

- We will see many fragments of programs that look like this:

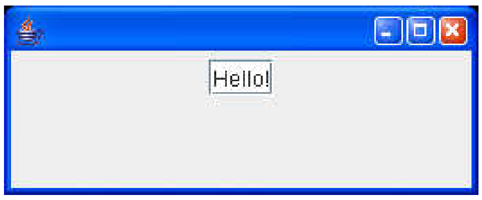
  `object name . action name (some information)`

  For example

  `System.exit(0)`
- The `object name` always indicates something, some part of the software, that has facilities or "knows" how to carry out various actions
- The `action name` indicates a facility made available by the named object, or an action that it "knows" how to do
  - It is called a *method* name
- The whole fragment is a *method call and* means that the named object is requested to carry out the named action
- The `some information` is extra details for the requested action
  - It is called the *parameter(s)* of the method call

Second simple application: Another way to display a text message (JFS, p17)

- The previous example had no "permanent window"
  - Not common!
- This example has a permanent window and displays a message in a *text field* in that window
  - Could be combined with pop-up dialogues too
- Need to set up a window
  - More administratively complex than the previous example
- Appearance, with no interesting behaviour except that the text field content can be altered:



Demo
Greeting

The Java source text (JFS, p18) – Slide 1/2

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Greeting extends JFrame {

    private JTextField textField;

    public static void main (String[] args) {
        Greeting frame = new Greeting();
        frame.setSize(300, 200);
        frame.createGUI();
        frame.setVisible(true);
    }

```

Note: NOT `frame.show();` (JFS4)

Slide 2/2

```
    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container window = getContentPane();
        window.setLayout(new FlowLayout() );
        textField = new JTextField("Hello!");
        window.add(textField);
    }
}
```

- No need to understand all this yet
  - It will be a standard framework for most examples
- Note all the **`object.action(…)`** method calls
- GUI = Graphical User Interface
- We have added a new *method* of our own: **`createGUI`**
  - It is *called* from **`main`**

End of Section