

List of Slides

- 1 Title
- 2 **Chapter 13:** Graphical user interfaces
- 3 Chapter aims
- 4 **Section 2:** Example: Hello world with a GUI
- 5 Aim
- 6 Hello world with a GUI
- 7 Hello world with a GUI
- 8 Package: `java.awt` and `javax.swing`
- 9 Hello world with a GUI
- 10 GUI API: `JFrame`
- 11 Hello world with a GUI
- 12 Class: extending another class
- 13 Hello world with a GUI
- 14 GUI API: `JFrame: setTitle()`
- 15 Hello world with a GUI
- 16 GUI API: `JFrame: getContentPane()`

17 GUI API: Container
18 Hello world with a GUI
19 GUI API: JLabel
20 GUI API: Container: add()
21 Hello world with a GUI
22 GUI API: JFrame: setDefaultCloseOperation()
24 Hello world with a GUI
25 GUI API: JFrame: pack()
26 Hello world with a GUI
27 GUI API: JFrame: setVisible()
28 Hello world with a GUI
29 The full HelloWorld code
30 Trying it
31 Coursework: HelloWorld GUI in French
32 **Section 3:** Example:Hello solar system with a GUI
33 Aim
34 Hello solar system with a GUI
35 GUI API: LayoutManager

36 GUI API: LayoutManager: FlowLayout
37 Hello solar system with a GUI
38 GUI API: Container: setLayout()
39 Hello solar system with a GUI
40 Hello solar system with a GUI
41 Trying it
42 Trying it
43 Coursework: HelloFamily GUI
44 **Section 4:** Example:Hello solar system with a GridLayout
45 Aim
46 Hello solar system with a GridLayout
47 GUI API: LayoutManager: GridLayout
49 Hello solar system with a GridLayout
50 Hello solar system with a GridLayout
51 Hello solar system with a GridLayout
52 Trying it
53 Coursework: HelloFamily GUI with GridLayout
54 **Section 5:** Adding JLabels in a loop

55 Aim
56 Adding JLabels in a loop
57 Coursework: TimesTable using JLabels
58 **Section 6:** Example:Tossing a coin
59 Aim
60 Tossing a coin
61 Execution: parallel execution – threads
63 Tossing a coin
64 Execution: parallel execution – threads: the GUI event thread
65 Execution: event driven programming
66 Tossing a coin
67 GUI API: Listeners
72 Tossing a coin
73 GUI API: JButton
74 Tossing a coin
75 Tossing a coin
76 GUI API: JButton: addActionListener()
77 Tossing a coin

78 Tossing a coin
79 The TossCoinActionListener class
80 Interface
81 GUI API: Listeners: ActionListener interface
82 The TossCoinActionListener class
84 GUI API: ActionEvent
85 GUI API: Listeners: ActionListener interface: actionPerformed()
86 The TossCoinActionListener class
87 GUI API: JLabel: setText()
88 The TossCoinActionListener class
89 The **event driven process**
90 Trying it
91 **Section 7:** Example:Stop clock
92 Aim
93 Stop clock
94 Stop clock
95 Stop clock
96 Stop clock

97 Stop clock
98 Stop clock
99 Stop clock
100 Standard API: `System.currentTimeMillis()`
101 Stop clock
103 Stop clock
104 Trying it
105 Trying it
106 Coursework: `StopClock` with split time
107 **Section 8:** Example: `GCD` with a GUI
108 Aim
109 `GCD` with a GUI
110 The `MyMath` class
111 The `MyMath` class
112 The `GCD` class
113 GUI API: `JTextField`
114 The `GCD` class
115 The `GCD` class

117 The GCD class
118 GUI API: JTextField: getText()
119 GUI API: JTextField: setText()
120 The GCD class
121 The GCD class
122 Trying it
123 Coursework: GCD GUI for three numbers
124 **Section 9:** Enabling and disabling components
125 Aim
126 Enabling and disabling components
127 GUI API: JButton: setEnabled()
128 GUI API: JTextField: setEnabled()
129 GUI API: JButton: setText()
130 Coursework: StopClock using a text field and disabled split button
131 **Section 10:** Example:Single times table with a GUI
132 Aim
133 Single times table with a GUI
134 GUI API: JTextArea

135 Single times table with a GUI
136 GUI API: `LayoutManager: BorderLayout`
138 Single times table with a GUI
140 GUI API: `Container: add()`: adding with a position constraint
141 Single times table with a GUI
142 Single times table with a GUI
143 GUI API: `JTextArea: setText()`
144 GUI API: `JTextArea: append()`
145 Single times table with a GUI
146 Single times table with a GUI
147 Single times table with a GUI
148 Trying it
149 Trying it
150 Trying it
151 **Section 11:** Example:GCD with Panels
152 Aim
153 GCD with Panels
154 GUI API: `JPanel`

155 GCD with Panels
156 GCD with Panels
157 GCD with Panels
158 GCD with Panels
160 GCD with Panels
161 GCD with Panels
162 GCD with Panels
163 GCD with Panels
164 GCD with Panels
165 GCD with Panels
166 GCD with Panels
167 Trying it
168 Trying it
169 Trying it
170 **Section 12:** Example:Single times table with a JScrollPane
171 Aim
172 Single times table with a JScrollPane
173 GUI API: JScrollPane

174 Single times table with a ScrollPane
175 GUI API: JTextField: initial value
176 Single times table with a ScrollPane
177 Single times table with a ScrollPane
179 Single times table with a ScrollPane
180 Single times table with a ScrollPane
181 Trying it
182 Trying it
183 Trying it
184 Coursework: ThreeWeights GUI
185 **Section 13:** Example:Age history with a GUI
186 Aim
187 Age history with a GUI
188 Age history with a GUI
189 Age history with a GUI
190 Age history with a GUI
191 Age history with a GUI
192 Age history with a GUI

193 GUI API: `LayoutManager: FlowLayout: alignment`
194 Age history with a GUI
195 Age history with a GUI
196 Age history with a GUI
197 Age history with a GUI
198 Age history with a GUI
199 Age history with a GUI
200 GUI API: `ActionEvent: getSource()`
201 Age history with a GUI
203 Age history with a GUI
204 Trying it
205 Trying it
206 Trying it
207 Trying it
208 Concepts covered in this chapter

Java Just in Time

John Latham

November 27, 2018

Chapter 13

Graphical user interfaces

Chapter aims

- We explore Java technology required to have **graphical user interfaces (GUIs)**.
- Introduce significant number of **classes** from Java **API**
 - dedicated to providing parts of GUIs.
- Also discuss **event driven programming**
 - relies on notion of **threads** to achieve parallel execution.

Section 2

Example:

Hello world with a GUI

Aim

AIM: To give a first introduction to Java **graphical user interface (GUI)** programs, in particular, the **classes** `JFrame`, `Container` and `JLabel`, together with the `java.awt` and `javax.swing` **packages** they belong to. We also talk about the idea of a class **extending** another class.

Hello world with a GUI

- All our programs so far have used
 - **command line arguments**
 - **standard output**
 - **textual user interface**
 - * via Scanner on **standard input**.
- Most modern **application programs** aimed at users who expect to interact via **graphical user interface** or **GUI**.
- First program displays single message in new window on screen
 - vehicle for introducing many concepts needed in more sophisticated GUIs.

Hello world with a GUI

- GUI programs in Java use **Java Swing** system
 - built on top of older **Abstract Windowing Toolkit**.
- Number of standard **classes** offering GUI features
 - we make **instances** and plug together to make GUIs.
- These standard classes grouped into **packages**.

Package: `java.awt` and `javax.swing`

- Inside **package** group `java` is package `awt`
 - full name `java.awt`.
- Contains **classes** for original Java **graphical user interface**
 - **Abstract Windowing Toolkit (AWT)**.
- E.g. class inside `java.awt` called `Container`
 - **fully qualified name** `java.awt.Container`.
- Package group `javax` contains package called `swing`
 - classes which make up more modern **Java Swing** system
 - * built on top of AWT.
- E.g. class inside `javax.swing` called `JFrame`
 - fully qualified name `javax.swing.JFrame`.
- Java **GUI** programs typically need classes from both these packages.

Hello world with a GUI

- Our HelloWorld program use three GUI classes.

```
001: import java.awt.Container;
```

```
002: import javax.swing.JFrame;
```

```
003: import javax.swing.JLabel;
```

- Window our program makes appear is instance of `javax.swing.JFrame`
 - just `JFrame` (thanks to `import`).

- Each **instance** of `javax.swing.JFrame` corresponds to a window on screen.

Hello world with a GUI

- JFrame represents basic empty windows.
- Our program has additional logic to make non-empty window
 - create class which is **extension** of JFrame.

Class: extending another class

- A **class** may **extend** another
 - use **reserved word extends**.
- E.g. HelloWorld extends javax.swing.JFrame.

```
import javax.swing.JFrame;

public class HelloWorld extends JFrame
```
- All **instances** of HelloWorld have JFrame properties
 - but also properties defined in HelloWorld.
- Adding new properties
 - without changing original class
 - * new class is **extension** of it.
- A HelloWorld object is a JFrame object
 - with extra stuff.

Hello world with a GUI

```
005: // Program to display a Hello World greeting in a window.  
006: public class HelloWorld extends JFrame  
007: {
```

- Set window title in **constructor method**....

GUI API: JFrame: setTitle()

- javax.swing.JFrame has **instance method** setTitle()
 - takes a String – use as window title
 - typically appears in title bar.

- Because HelloWorld is extension of JFrame
 - automatically contains setTitle() instance method
 - so can use it without prefix.

```
008: // Constructor.  
009: public HelloWorld()  
010: {  
011:     setTitle("Hello World");
```

- Next add GUI component to **content pane** of window....

GUI API: JFrame: getContentPane()

- `javax.swing.JFrame` has **instance method** `getContentPane()`
 - **returns content pane** of the `JFrame`
 - part that holds/contains **GUI** components of the window
 - is **instance** of `java.awt.Container`.

- `java.awt.Container` implements part of a **GUI**
 - each **instance** is a component allowed to contain other components.

Hello world with a GUI

- Thanks to import
 - refer to `java.awt.Container` as just `Container`.
- Also `getContentPane()` automatically part of extension to `JFrame`
 - can use it without prefix.

```
012:     Container contents = getContentPane();
```

- Wish to add `JLabel` to content pane....

- `javax.swing.JLabel` implements part of a **GUI**
 - displays small piece of text
 - label is **method argument** to **constructor method**.

GUI API: Container: add()

- `java.awt.Container` has **instance method** `add()`
 - takes **GUI** component
 - includes it in components to be displayed in container.

Hello world with a GUI

```
013:     contents.add(new JLabel("Greetings to all who dwell on Planet Earth!"));
```

- Next: what should happen when user presses close button on title bar?....

- `javax.swing.JFrame` has **instance method** `setDefaultCloseOperation()`
 - has **method parameter**: what should happen when user presses close button on window title bar.
 - Four possible values:
 - * **Do nothing on close** – Don't do anything.
 - * **Hide on close** – Hide the window, so that it is no longer visible, but do not destroy it.
 - * **Dispose on close** – Destroy the window.
 - * **Exit on close** – Exit the whole program.

- Argument is actually an `int`
 - do not need to know exact value
 - four **class constants** to use.

```
public static final int DO_NOTHING_ON_CLOSE = ?;
```

```
public static final int HIDE_ON_CLOSE = ?;
```

```
public static final int DISPOSE_ON_CLOSE = ?;
```

```
public static final int EXIT_ON_CLOSE = ?;
```

- Use whichever constant suits us, e.g:

```
setDefaultCloseOperation(DISPOSE_ON_CLOSE);
```

Hello world with a GUI

```
014:      setDefaultCloseOperation(EXIT_ON_CLOSE);
```

- Finally: make `JFrame` **pack** itself....

GUI API: JFrame: pack ()

- `javax.swing.JFrame` has **instance method** `pack ()`
 - makes `JFrame` arrange itself ready for appearing on screen
 - * works out sizes / positions of all components
 - * size of the window itself.
- Typically `pack ()` called after all **GUI** components been added.

Hello world with a GUI

```
015:     pack();  
016: } // HelloWorld
```

- Have **main method** create instance of HelloWorld.

```
019: // Create a HelloWorld and make it appear on screen.  
020: public static void main(String[] args)  
021: {  
022:     HelloWorld theHelloWorld = new HelloWorld();
```

- And show it on screen....

GUI API: JFrame: setVisible()

- javax.swing.JFrame has **instance method** setVisible()
 - has **boolean method parameter**
 - * **true**: JFrame **object** causes window to appear on physical screen
 - * **false**: – disappear (hide).

Hello world with a GUI

```
023:     theHelloWorld.setVisible(true);  
024: } // main  
025:  
026: } // class HelloWorld
```

The full HelloWorld code

```
001: import java.awt.Container;
002: import javax.swing.JFrame;
003: import javax.swing.JLabel;
004:
005: // Program to display a Hello World greeting in a window.
006: public class HelloWorld extends JFrame
007: {
008:     // Constructor.
009:     public HelloWorld()
010:     {
011:         setTitle("Hello World");
012:         Container contents = getContentPane();
013:         contents.add(new JLabel("Greetings to all who dwell on Planet Earth!"));
014:         setDefaultCloseOperation(EXIT_ON_CLOSE);
015:         pack();
016:     } // HelloWorld
017:
018:
019:     // Create a HelloWorld and make it appear on screen.
020:     public static void main(String[] args)
021:     {
022:         HelloWorld theHelloWorld = new HelloWorld();
023:         theHelloWorld.setVisible(true);
024:     } // main
025:
026: } // class HelloWorld
```


Trying it

Console Input / Output

```
$ javac HelloWorld.java
$ java HelloWorld
# The GUI appears, and then when we press the close button on the window frame,
# the program ends.
$ _
```

Run



Coffee time: Suppose we wanted the program to display *two* windows giving the greeting, instead of one. What changes would we need to make?

(Summary only)

Write a **GUI** program to greet the world, in French.

Section 3

Example:

Hello solar system with a GUI

AIM: To introduce the notion of **layout manager** and, in particular, `FlowLayout`.

Hello solar system with a GUI

```
001: import java.awt.Container;
002: import java.awt.FlowLayout;
003: import javax.swing.JFrame;
004: import javax.swing.JLabel;
005:
006: // Program to display a greeting to all nine planets, in a window.
007: public class HelloSolarSystem extends JFrame
008: {
009:     // Constructor.
010:     public HelloSolarSystem()
011:     {
012:         setTitle("Hello Solar System");
013:         Container contents = getContentPane();
```

- Nine JLabel **objects**: shall use a FlowLayout....

- A **layout manager** is **class**
 - with logic for laying out **GUI** components in a `java.awt.Container`.
- Various types, including:
 - `java.awt.FlowLayout` – arrange the components in a horizontal line.
 - `java.awt.GridLayout` – arrange the components in a grid.
 - `java.awt.BorderLayout` – arrange the components with one at the centre, and one at each of the four sides.

- `java.awt.FlowLayout` is **layout manager**
 - positions components in a `java.awt.Container` in horizontal row
 - * appear in order were added to the container.

Hello solar system with a GUI

- Make **instance** of layout manager
- pass to Container via `setLayout()`...

GUI API: Container: `setLayout()`

- `java.awt.Container` has **instance method** `setLayout()`
 - takes **instance** of a **layout manager**
 - uses it to lay out components in that container
 - * when window is **packed**.

Hello solar system with a GUI

```
015:    // We want the planet names to appear in one line.
016:    contents.setLayout(new FlowLayout());
018:    contents.add(new JLabel("Hello Mercury!"));
019:    contents.add(new JLabel("Hello Venus!"));
020:    contents.add(new JLabel("Hello Earth!"));
021:    contents.add(new JLabel("Hello Mars!"));
022:    contents.add(new JLabel("Hello Jupiter!"));
023:    contents.add(new JLabel("Hello Saturn!"));
024:    contents.add(new JLabel("Hello Uranus!"));
025:    contents.add(new JLabel("Hello Neptune!"));
026:    contents.add(new JLabel("Goodbye Pluto!"));
027:
028:    setDefaultCloseOperation(EXIT_ON_CLOSE);
029:    pack();
030: } // HelloSolarSystem
```

Hello solar system with a GUI

- Then **main method**
 - creates instance
 - makes it visible.

```
033: // Create a HelloSolarSystem and make it appear on screen.
034: public static void main(String[] args)
035: {
036:     HelloSolarSystem theHelloSolarSystem = new HelloSolarSystem();
037:     theHelloSolarSystem.setVisible(true);
038: } // main
039:
040: } // class HelloSolarSystem
```

Trying it

Console Input / Output

```
$ javac HelloSolarSystem.java  
$ java HelloSolarSystem  
$ _
```

Run

Trying it



(Summary only)

Write a **GUI** program to greet your family.

Section 4

Example:

Hello solar system with a GridLayout

Aim

AIM: To introduce the **layout manager** called GridLayout.

Hello solar system with a GridLayout

- Another version of HelloSolarSystem
 - but use GridLayout...

- `java.awt.GridLayout` is **layout manager**
 - positions all components in a `java.awt.Container` in rectangular grid.
 - Container divided into equal-sized rectangles
 - one component placed in each rectangle.
 - Components appear in order were added to container
 - * filling one row at a time.
- Provide pair of **method arguments** to **constructor method**
 - first: number of rows
 - second: number of columns
 - one should be zero.

- E.g. 3 rows, as many columns as needed.

```
new GridLayout(3, 0);
```

- E.g. 2 columns, as many rows as needed.

```
new GridLayout(0, 2);
```

- If both non-zero, *columns is totally ignored!*
- Neither may be negative.
- At least one non-zero – else **run time error**.
- Can also provide horizontal / vertical gaps via another constructor.
 - E.g. 5 columns, horizontal gap = 10 pixels, vertical = 20 pixels.

```
new GridLayout(0, 5, 10, 20);
```

- Pixel is smallest unit of display position.

Hello solar system with a GridLayout

```
001: import java.awt.Container;
002: import java.awt.GridLayout;
003: import javax.swing.JFrame;
004: import javax.swing.JLabel;
005:
006: // Program to display a greeting to all nine planets, in a window.
007: public class HelloSolarSystem extends JFrame
008: {
009:     // Constructor.
010:     public HelloSolarSystem()
011:     {
012:         setTitle("Hello Solar System");
013:         Container contents = getContentPane();
```

- 9 (ex) planets: have 3 x 3 grid, 10 pixels between rows, 20 between columns.

Hello solar system with a GridLayout

```
015:    // Set layout to be a grid of 3 columns.
016:    // This will also give 3 rows, as there are 9 items.
017:    contents.setLayout(new GridLayout(0, 3, 20, 10));
018:
019:    contents.add(new JLabel("Hello Mercury!"));
020:    contents.add(new JLabel("Hello Venus!"));
021:    contents.add(new JLabel("Hello Earth!"));
022:    contents.add(new JLabel("Hello Mars!"));
023:    contents.add(new JLabel("Hello Jupiter!"));
024:    contents.add(new JLabel("Hello Saturn!"));
025:    contents.add(new JLabel("Hello Uranus!"));
026:    contents.add(new JLabel("Hello Neptune!"));
027:    contents.add(new JLabel("Goodbye Pluto!"));
028:
029:    setDefaultCloseOperation(EXIT_ON_CLOSE);
030:    pack();
031: } // HelloSolarSystem
```

Hello solar system with a GridLayout

- The only change was the **layout manager**.

```
034: // Create a HelloSolarSystem and make it appear on screen.
035: public static void main(String[] args)
036: {
037:     HelloSolarSystem theHelloSolarSystem = new HelloSolarSystem();
038:     theHelloSolarSystem.setVisible(true);
039: } // main
040:
041: } // class HelloSolarSystem
```

Trying it

Console Input / Output

```
$ javac HelloSolarSystem.java  
$ java HelloSolarSystem  
$ _
```

Run



(Summary only)

Write a **GUI** program to greet your family, using a GridLayout.

Section 5

Adding JLabels in a loop

Aim

AIM: To illustrate the idea of creating **graphical user interface (GUI)** components in a **loop**.

Adding JLabels in a loop

- No example here!
- Observation: a **GUI** can have variable number of components
 - can be created / added by a **loop**.

Coursework: TimeTable using JLabels

(Summary only)

Write a program to display a times table, using a **GUI** with `JLabel` **objects**.

Section 6

Example: Tossing a coin

Aim

AIM: To introduce the Java **listener** model together with JButton, ActionEvent and ActionListener. This requires some discussion of the notion of **threads** and **event driven programming**, as well as **interfaces**. We also re-visit JLabel.

Tossing a coin

- Coin tosser
 - single button
 - * each button click causes `heads` or `tails` displayed in a label.
- But first: **threads...**

Execution: parallel execution – threads

- Computers *appear* to perform several tasks at same time
 - e.g. we can run several programs at once.
- In **operating system** each program runs in separate **process**
 - **central processing unit** time shared between current processes.
- Java **virtual machine** also can do this: **threads**
 - allows single program to do many things ‘at the same time’.

- JVM creates **main thread** at start
 - uses it to **run** body of **main method**
 - **executes statements** in main method
 - * including **method calls**
 - at end of main method: thread terminates.
- JVM also ends if that was only thread.
- But program can create other threads
 - JVM (by default) continues running until all threads ended.

Tossing a coin

- Before this chapter have used only **main thread**
 - so JVM ended when **main method** reached end.
- GUI programs: more complex.

Execution: parallel execution – threads: the GUI event thread

- When **GUI** program first places window on screen
 - JVM creates **GUI event thread**.
- When **main thread** ends, program does *not* – good!
- **GUI event thread** mostly just sleeps waiting for user.
- When user does interesting thing
 - e.g. move mouse in window, click, type, etc.
- **operating system** informs JVM
 - JVM wakes up GUI event thread.
- GUI event thread checks whether interesting
 - if so, **executes** code designated to process that event.
 - E.g. user has pressed our GUI button
 - * we want specific thing to happen.

Execution: event driven programming

- Writing **GUI** programs is about constructing code to
 - set up the GUI
 - process **events** associated with the end user's actions.
- Known as **event driven programming**.
- Our **main method** sets up GUI, then ends.
- Then our code for processing GUI events does the work
 - when end user does things
 - program becomes driven by events.

Tossing a coin

- Each time button is pressed, we want coin to be tossed.
- Need to link **event** processing code (toss coin) to button
 - using a **listener**...

- Java uses **listener** model for processing **GUI events**.
- When user causes an event
 - e.g. presses a GUI button
- **GUI event thread** creates **object** representing event.
- This has **event source**: the GUI object that caused the event
 - e.g. the button the user pressed.
- Event sources (e.g. buttons) keep set of **listener** objects
 - that have been registered with it.
- GUI event thread processes event
 - calls particular **instance method** belonging to each registered listener.

- Abstract example: we have some object that can be an event source
 - e.g. a button, but let's keep it abstract.

```
SomeKindOfEventSource source = new SomeKindOfEventSource(...);
```

- Wish events to be processed by some particular code
 - put that in **class** called `SomeKindOfEventListener`.

```
public class SomeKindOfEventListener
{
    public void processSomeKindOfEvent(SomeKindOfEvent e)
    {
        ... Code that deals with the event.
    } // processSomeKindOfEvent
} // class SomeKindOfEventListener
```

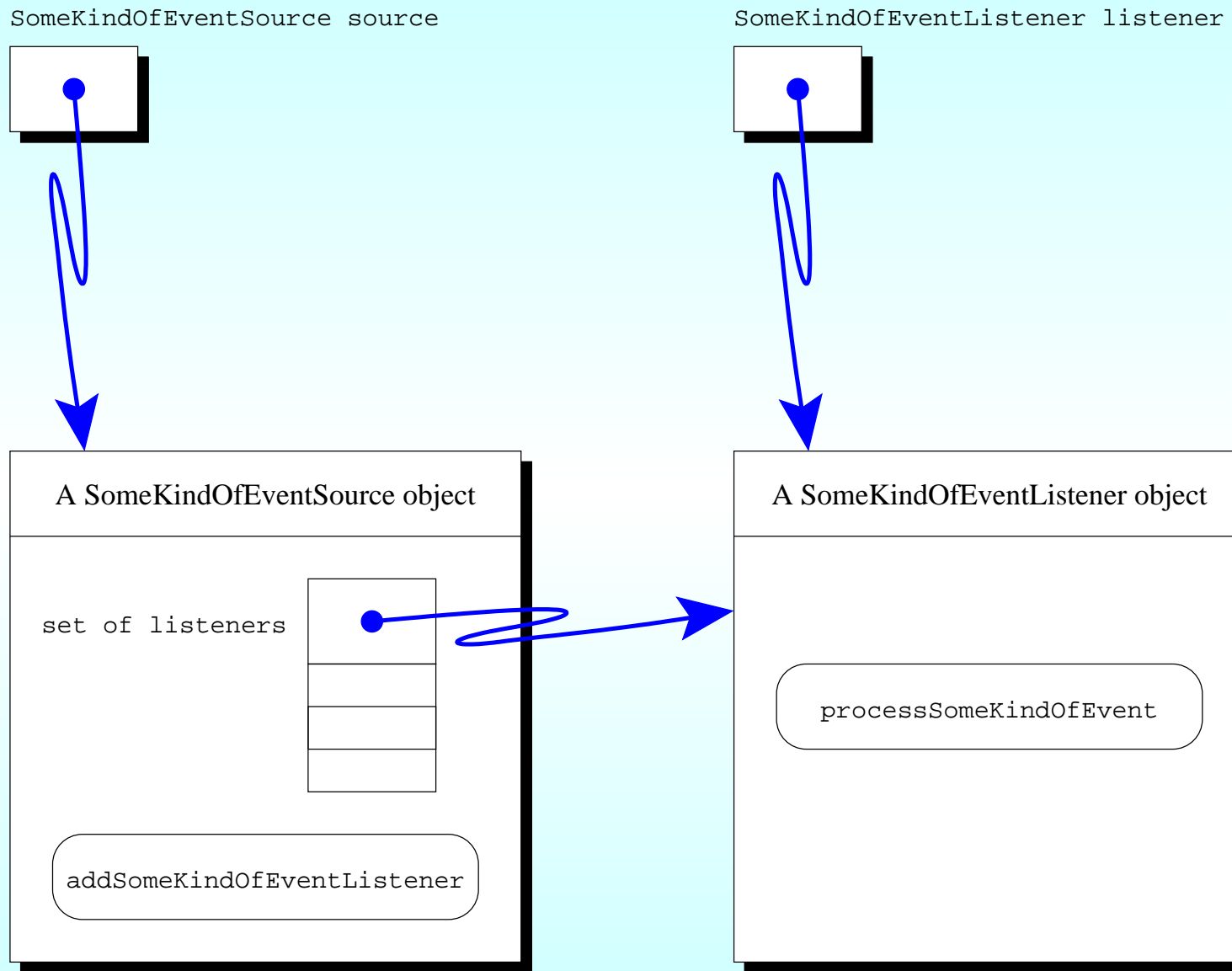
- Link code to event
 - make instance of `SomeKindOfEventListener`
 - register it with event source.

```
SomeKindOfEventListener listener = new SomeKindOfEventListener(...);
```

```
source.addSomeKindOfListener(listener);
```

- This code would run in **main thread** during GUI set up.
- Diagram shows finished set up...

GUI API: Listeners



- When event happens
 - GUI event thread looks at set of listeners in source object
 - call `processSomeKindOfEvent()` belonging to each of them.
- Our source object generates event
 - `processSomeKindOfEvent()` in our listener is called.
- Java Swing has several different kinds of listener
 - for different kinds of event.
- Above example is **abstraction** only!
- E.g. events generated by buttons
 - known as `ActionEvents`
 - processed by `ActionListener` objects
 - * which have `actionPerformed()` instance method
 - linked to event source by `addActionListener()`.

Tossing a coin

```
001: import java.awt.Container;
002: import java.awt.GridLayout;
003: import javax.swing.JButton;
004: import javax.swing.JFrame;
005: import javax.swing.JLabel;
006:
007: // A simple coin tossing program. The button tosses the coin.
008: // The label shows how many tosses there have been
009: // and whether the latest one was heads or tails.
010: public class CoinTosser extends JFrame
011: {
```

- Constructor will create and add a `JLabel` to **content pane**
 - and also a `JButton`...

GUI API: JButton

- `javax.swing.JButton` implements **GUI** component
 - providing button for the end user to 'press' using mouse.
- Button label is `String` given to `JButton` **constructor method**.

Tossing a coin

```
012: // Constructor.
013: public CoinTosser()
014: {
015:     setTitle("Coin Tosser");
016:     Container contents = getContentPane();
017:     // Use a grid layout with one column.
018:     contents.setLayout(new GridLayout(0, 1));
019:
020:     JLabel headsOrTailsJLabel = new JLabel("Not yet tossed");
021:     contents.add(headsOrTailsJLabel);
022:
023:     JButton tossCoinJButton = new JButton("Toss the Coin");
024:     contents.add(tossCoinJButton);
```

Tossing a coin

- We will write separate class called `TossCoinActionListener`
 - will be a **listener** for the `JButton`.
- To work, must ensure that `TossCoinActionListener` is an `ActionListener`
 - kind of listener that can be linked to `JButtons`.
- Here we create **instance** of `TossCoinActionListener`
 - and register it as `ActionListener` for our button...

- javax.swing.JButton has **instance method** addActionListener()
 - takes ActionListener **object**
 - remembers that **listener** interested in events from this button.

```
public void addActionListener(ActionListener listener)
{
    ... Remember that listener wants to be informed of action events.
} // addActionListener
```

Tossing a coin

```
026:    // The action listener for the button needs to update the heads/tails
027:    // JLabel, so we pass that reference to its constructor.
028:    TossCoinActionListener listener
029:        = new TossCoinActionListener(headsOrTailsJLabel);
030:    tossCoinJButton.addActionListener(listener);
031:
032:    setDefaultCloseOperation(EXIT_ON_CLOSE);
033:    pack();
034: } // CoinTosser
```


Tossing a coin

- Our **main method**: similar to previous examples.

```
037: // Create a CoinTosser and make it appear on screen.
038: public static void main(String[] args)
039: {
040:     CoinTosser theCoinTosser = new CoinTosser();
041:     theCoinTosser.setVisible(true);
042: } // main
043:
044: } // class CoinTosser
```

The TossCoinActionListener class

- Pressing a button causes an `ActionEvent`
 - **listener** must **implement** `ActionListener` **interface**...

Interface

- An **interface**: like a **class** except all **instance methods** have no bodies!
- Used as basis of kind of contract
 - some class declares it **implements** an interface
 - * thus provides bodies for instance methods listed in interface.

- E.g.

```
public class MyClass implements SomeInterface
{
    ... implementation for every instance method listed in SomeInterface
} // MyClass
```

- Compiler checks `MyClass` implements all methods listed in `SomeInterface`.
- E.g. if some method has **method parameter** of **type** `SomeInterface`
 - **instance** of `MyClass` could be supplied as **method argument**.

- Standard **interface** `java.awt.event.ActionListener`
 - contains body-less **instance method** `actionPerformed`.
- Idea: full implementation contains specific code required to process **event**
 - caused by the user pressing a **GUI** button (etc.).

The TossCoinActionListener class

- Declare that `TossCoinActionListener` implements `ActionListener`.
- Then provide full implementation of **instance method** listed in `ActionListener`.

```
001: import java.awt.event.ActionEvent;
002: import java.awt.event.ActionListener;
003: import javax.swing.JLabel;
004:
005: // The ActionListener for CoinTosser's TossCoin JButton. Each time
006: // actionPerformed is called, we count the number of tosses, and update the
007: // given JLabel with that count, plus either "Heads" or "Tails".
008: public class TossCoinActionListener implements ActionListener
009: {
```

The TossCoinActionListener class

```
010: // The JLabel that needs to be updated.
011: private final JLabel headsOrTailsJLabel;
012:
013: // We count the tosses, so it is clear when we have a new toss.
014: private int noOfTossesSoFar = 0;
015:
016:
017: // Constructor.
018: public TossCoinActionListener(JLabel requiredHeadsOrTailsJLabel)
019: {
020:     headsOrTailsJLabel = requiredHeadsOrTailsJLabel;
021: } // TossCoinActionListener
```

- Next is `actionPerformed()`...

- When end user performs 'action' (e.g. pressing a button)
 - **GUI event thread** creates **instance** of `java.awt.event.ActionEvent`
 - stores **reference** to **event source object**
 - * e.g. the button that was pressed.

GUI API: Listeners: ActionListener interface: `actionPerformed()`

- After creating `java.awt.event.ActionEvent`
 - **GUI event thread** finds from **event source** which `ActionListener` **objects** have registered with it.
- GUI event thread then calls `actionPerformed` belonging to each one
 - passing the `ActionEvent` as **method argument**.
- Heading of `actionPerformed()` is:

```
public void actionPerformed(ActionEvent event)
```
- Each implementation performs whatever task is needed as response to action
 - e.g. different buttons, different effects.

The TossCoinActionListener class

- Here we do not need to look at `ActionEvent` because have only one button!
 - If had more than one, then could determine which from the **event source** inside the `ActionEvent`.
- Our code needs to change text of `JLabel1...`

GUI API: JLabel: setText()

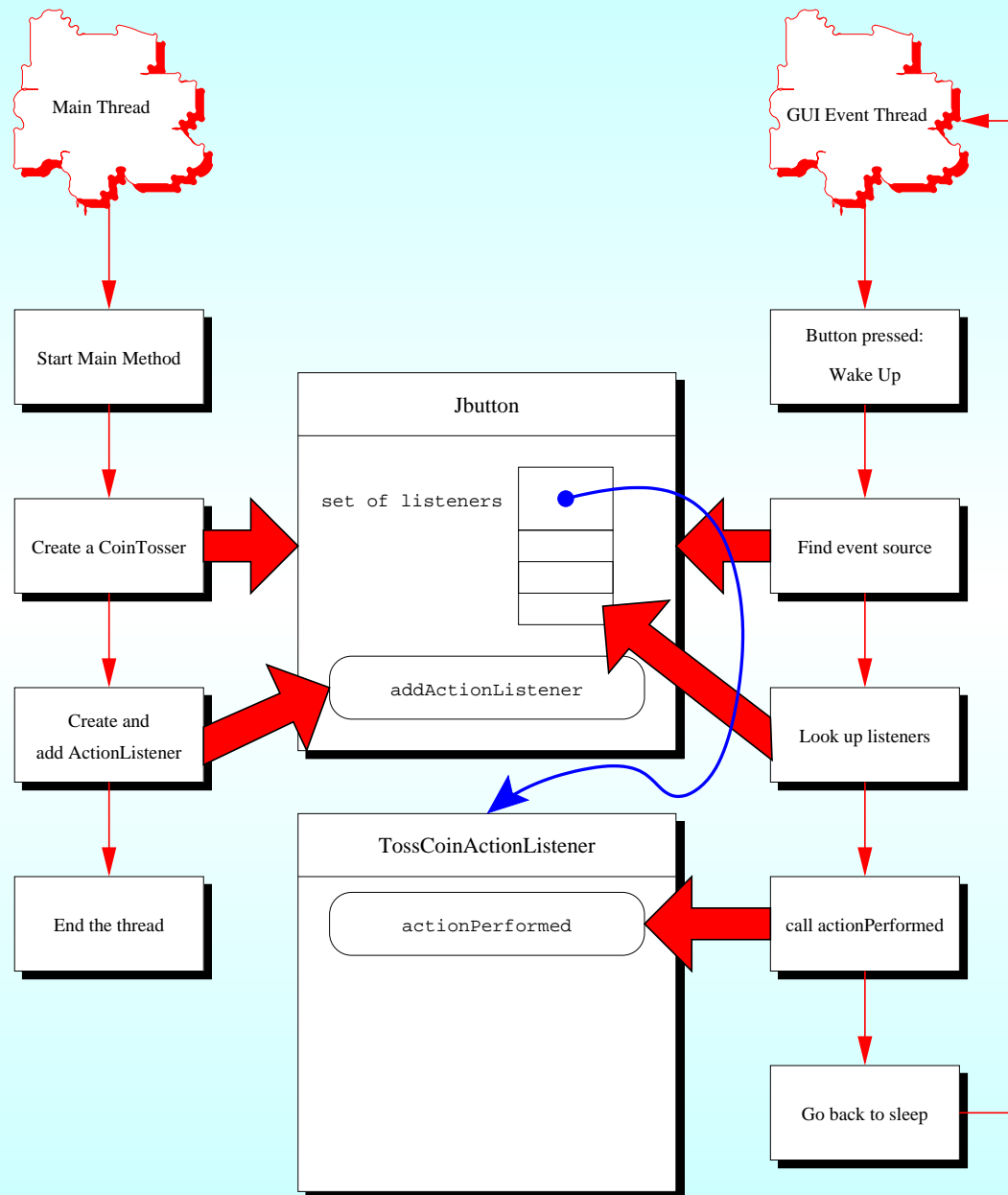
- `javax.swing.JLabel` has **instance method** `setText()`
 - changes text of label to given `String`.

The TossCoinActionListener class

```
024: // Action performed: update noOfTossesSoFar and headsOrTailsJLabel.
025: public void actionPerformed(ActionEvent event)
026: {
027:     noOfTossesSoFar++;
028:     if (Math.random() >= 0.5)
029:         headsOrTailsJLabel.setText("Toss " + noOfTossesSoFar + ": Heads");
030:     else
031:         headsOrTailsJLabel.setText("Toss " + noOfTossesSoFar + ": Tails");
032: } // actionPerformed
033:
034: } // class TossCoinActionListener
```

- A diagram to help?...

The event driven process

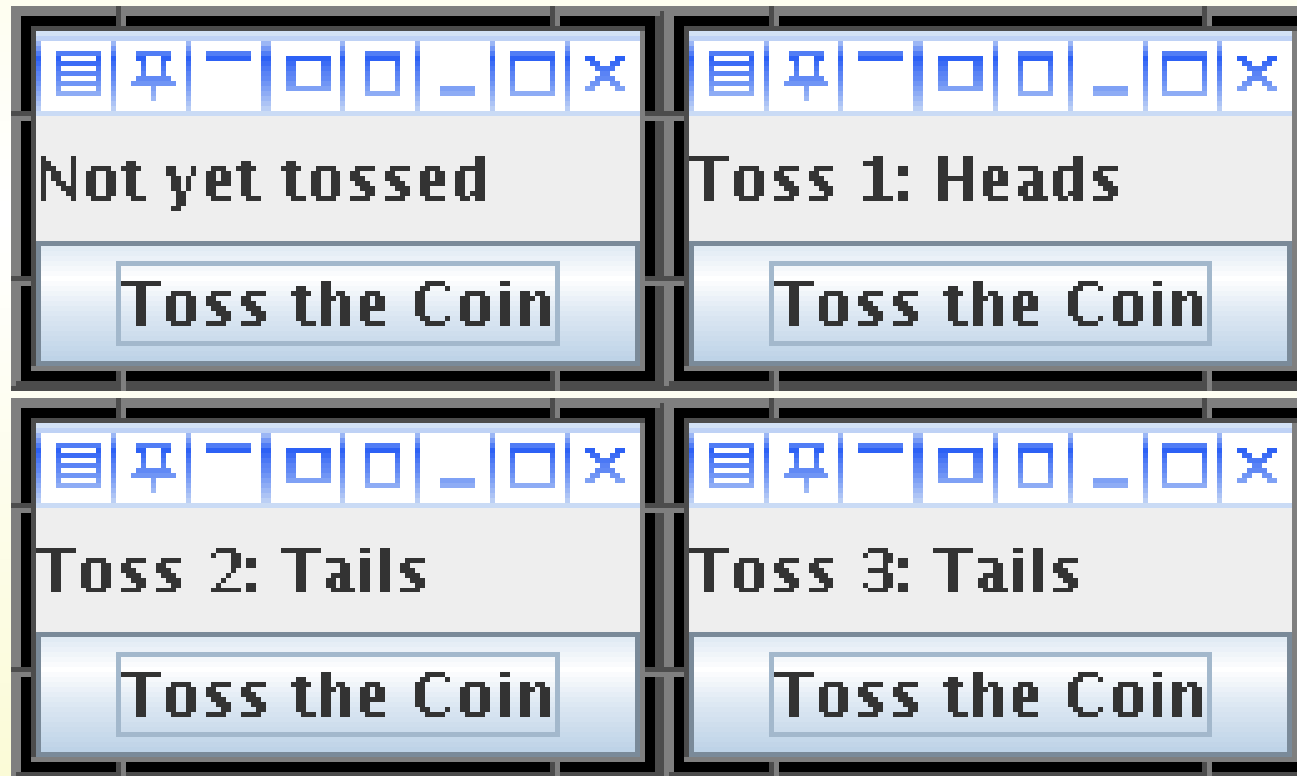


Trying it

Console Input / Output

```
$ javac CoinTosser.java  
$ java CoinTosser  
$ _
```

Run



Section 7

Example: Stop clock

AIM: To reinforce the Java **listener** model together with JButton, ActionEvent and ActionListener. We also introduce the idea of having the ActionListener **object** be the JFrame itself, and meet System.currentTimeMillis().

Stop clock

- Provide simple stop clock
 - single button toggles state of clock running and stopped
 - when started shows status
 - when stopped updates status and shows elapsed time.

```
001: import java.awt.Container;  
002: import java.awt.GridLayout;  
003: import java.awt.event.ActionEvent;  
004: import java.awt.event.ActionListener;  
005: import javax.swing.JButton;  
006: import javax.swing.JFrame;  
007: import javax.swing.JLabel;  
008:  
009: // A simple stop clock program. The button stops and starts the clock.  
010: // The clock records start time, stop time and shows elapsed time.
```


Stop clock

- Our **GUI** has start/stop button
 - want program to run some code when pressed
 - need `ActionListener` to process `ActionEvent` **objects**
 - but this time not use a separate class!
- We declare `StopClock` **extends** `JFrame`
 - and *also* **implements** `ActionListener`!

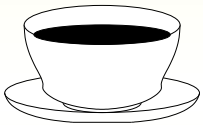
```
011: public class StopClock extends JFrame implements ActionListener
012: {
```

Stop clock

```
013: // True if and only if the clock is running.
```

```
014: private boolean isRunning = false;
```

- Java represents current time as number of milliseconds since midnight, January 1, 1970.



Coffee time: Find out, for example by searching on the Internet, what is the significance of midnight at the start of January 1st 1970.

```
016: // The time when the clock is started
```

```
017: // as milliseconds since midnight, January 1, 1970.
```

```
018: private long startTime = 0;
```

```
019:
```

```
020: // The time when the clock is stopped
```

```
021: // as milliseconds since midnight, January 1, 1970.
```

```
022: private long stopTime = 0;
```

Stop clock

- References to JLabel **objects** that will be updated.

```
024: // A label for showing the start time.
025: private final JLabel startTimeJLabel = new JLabel("Not started");
026:
027: // A label for showing the stop time.
028: private final JLabel stopTimeJLabel = new JLabel("Not started");
029:
030: // A label for showing the elapsed time.
031: private final JLabel elapsedTimeJLabel = new JLabel("Not started");
```

Coffee time: Declaring the above instance variables as **final variables** means their value cannot be changed. Why will that not stop us from changing the value (i.e. the text) of JLabels **referenced** by them?



Stop clock

```
034: // Constructor.
035: public StopClock()
036: {
037:     setTitle("Stop Clock");
038:
039:     Container contents = getContentPane();
040:     // Use a grid layout with one column.
041:     contents.setLayout(new GridLayout(0, 1));
042:
043:     contents.add(new JLabel("Started at:"));
044:     contents.add(startTimeJLabel);
045:
046:     contents.add(new JLabel("Stopped at:"));
047:     contents.add(stopTimeJLabel);
048:
049:     contents.add(new JLabel("Elapsed time (seconds):"));
050:     contents.add(elapsedTimeJLabel);
```

Stop clock

```
052:     JButton startStopJButton = new JButton("Start / Stop");
```

- This object is `ActionListener` for its own button
 - recall **this reference**: `this`.

```
053:     startStopJButton.addActionListener(this);
```

```
054:     contents.add(startStopJButton);
```

```
055:
```

```
056:     setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
057:     pack();
```

```
058: } // StopClock
```



Coffee Before continuing, make a sketch of what you think the
time: StopClock GUI will look like.

Stop clock

- Now `actionPerformed()`
 - called when user presses button.

```
061: // Perform action when the button is pressed.
062: public void actionPerformed(ActionEvent event)
063: {
064:     if (!isRunning)
065:     {
066:         // Start the clock.
```

- Get current time...

- `java.lang.System` contains **class method** `currentTimeMillis()`
 - **returns** current date and time as number of milliseconds since midnight, January 1, 1970.
 - `long` value.

Stop clock

- Note use of "" + startTime
 - is no version of setText() that takes an `int`.

```
067:     startTime = System.currentTimeMillis();
068:     startTimeJLabel.setText("" + startTime);
069:     stopTimeJLabel.setText("Running...");
070:     elapsedTimeJLabel.setText("Running...");
071:     isRunning = true;
072: } // if
```


Stop clock

```
073:     else // isRunning
074:     {
075:         // Stop the clock and show the updated times.
076:         stopTime = System.currentTimeMillis();
077:         stopTimeJLabel.setText("" + stopTime);
078:         long elapsedMilliseconds = stopTime - startTime;
079:         elapsedTimeJLabel.setText("" + elapsedMilliseconds / 1000.0);
080:         isRunning = false;
081:     } // else
082:     // It is a good idea to pack again
083:     // because the size of the labels may have changed.
084:     pack();
085: } // actionPerformed
```



Coffee Why do we divide the elapsed time by 1000?
time:

Stop clock



































































```
088: // Create a StopClock and make it appear on screen.
089: public static void main(String[] args)
090: {
091:     StopClock theStopClock = new StopClock();
092:     theStopClock.setVisible(true);
093: } // main
094:
095: } // class StopClock
```

Trying it

Console Input / Output

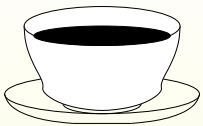
```
$ java StopClock  
$ _
```

Run

                                Started at: Not started Stopped at: Not started Elapsed time (seconds): Not started Start / Stop	                      Started at: 1562004720000 Stopped at: Running... Elapsed time (seconds): Running... Start / Stop	            Started at: 1562004720000 Stopped at: 1562004730009 Elapsed time (seconds): 10.009 Start / Stop
--	---	--

Trying it

Coffee time: Showing the start and stop time as milliseconds is, perhaps, interesting for someone new to that idea, but not really so for anyone who actually wants to use the program. Think about changing the **design** so that the GUI does not show the start and stop times, but just shows the running status (not-started, running or stopped) instead.



(Summary only)

Modify a stop clock program so that it has a split time button.

Section 8

Example: GCD with a GUI

Aim

AIM: To introduce JTextField.

GCD with a GUI

- Program to display the GCD of two numbers supplied by user.
- Place code for calculating GCD in separate **class**
 - can be easily reused in many programs.

The MyMath class

- Our **class** `MyMath` contains only `greatestCommonDivisor()`
 - at the moment...
- Intended for reuse so write **doc comments** for it.
- No need to present code here
 - only need documentation for how to use it!

Web Browser Window

greatestCommonDivisor

```
public static int greatestCommonDivisor(int multiple1OfGCD,  
                                          int multiple2OfGCD)
```

Computes the greatest common divisor of two numbers. The numbers must be positive.

Parameters:

multiple1OfGCD - One of the numbers.
multiple2OfGCD - The other number.

Returns:

The GCD of multiple1OfGCD and multiple2OfGCD.

Run

The GCD class

- GCD class provides GUI for program and **main method**.
- Use `javax.swing.JTextField` to get inputs...

```
001: import java.awt.Container;  
002: import java.awt.GridLayout;  
003: import java.awt.event.ActionEvent;  
004: import java.awt.event.ActionListener;  
005: import javax.swing.JButton;  
006: import javax.swing.JFrame;  
007: import javax.swing.JLabel;  
008: import javax.swing.JTextField;  
009:  
010: // Calculates the GCD of two integers.  
011: public class GCD extends JFrame implements ActionListener  
012: {
```

- `javax.swing.JTextField` implements part of **GUI**
 - allows user to enter small piece of text.
- One **constructor method** takes `int` **method parameter**
 - minimum number of **characters** wide enough to display.
- `JTextField` can also be used to display text generated from within program.

The GCD class

```
013: // A JTextField for each number.  
014: private final JTextField number1JTextField = new JTextField(20);  
015: private final JTextField number2JTextField = new JTextField(20);  
016:  
017: // A JTextField for the result.  
018: private final JTextField resultJTextField = new JTextField(20);
```

The GCD class

```
021: // Constructor.
022: public GCD()
023: {
024:     setTitle("GCD");
025:
026:     Container contents = getContentPane();
027:     contents.setLayout(new GridLayout(0, 1)); // Single column.
028:
029:     contents.add(new JLabel("Number 1"));
030:     contents.add(number1JTextField);
031:     contents.add(new JLabel("Number 2"));
032:     contents.add(number2JTextField);
033:
```

The GCD class

```
034:     JButton computeJButton = new JButton("Compute");
035:     contents.add(computeJButton);
036:     computeJButton.addActionListener(this);
037:
038:     contents.add(new JLabel("GCD of Number 1 and Number 2"));
039:     contents.add(resultJTextField);
040:
041:     setDefaultCloseOperation(EXIT_ON_CLOSE);
042:     pack();
043: } // GCD
```



Coffee time: Before continuing, make a sketch of what you think the GCD GUI will look like.

The GCD class

- `actionPerformed()` **instance method** will
 - get **data** from the input text fields
 - calculate GCD
 - put result in output text field.

GUI API: JTextField: getText()

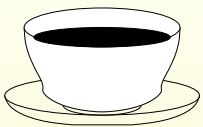
- `javax.swing.JTextField` has **instance method** `getText()`
 - no **method arguments**
 - **returns** `String` text contents of text field.

GUI API: JTextField: setText()

- `javax.swing.JTextField` has **instance method** `setText()`
 - takes `String` **method argument**
 - changes text of text field to it.

The GCD class

```
046: // Act upon the button being pressed.
047: public void actionPerformed(ActionEvent event)
048: {
049:     int number1 = Integer.parseInt(number1JTextField.getText());
050:     int number2 = Integer.parseInt(number2JTextField.getText());
051:     int theGCD = MyMath.greatestCommonDivisor(number1, number2);
052:     resultJTextField.setText("" + theGCD);
053: } // actionPerformed
```



Coffee time: Is there anything to stop the user from changing the result, by typing directly into its text field?

The GCD class

```
056: // Create a GCD and make it appear on screen.
057: public static void main(String[] args)
058: {
059:     GCD theGCD = new GCD();
060:     theGCD.setVisible(true);
061: } // main
062:
063: } // class GCD
```

Coffee time: Do we need the **local variable** theGCD in the main method? What if the body was just the following?

```
new GCD().setVisible(true);
```

Would that work?

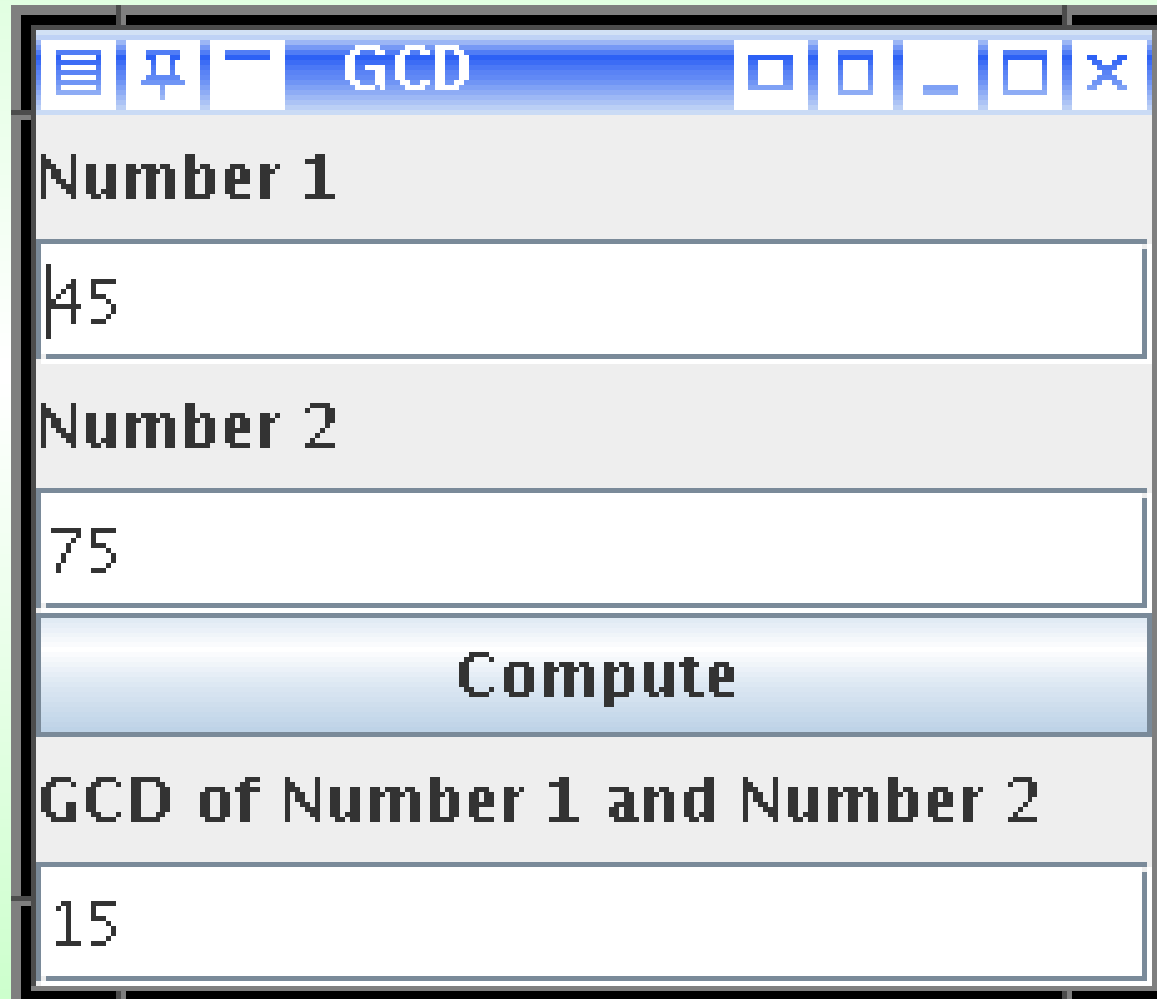


Trying it

Console Input / Output

```
$ java GCD  
$_
```

Run



Number 1
45

Number 2
75

Compute

GCD of Number 1 and Number 2
15

Coursework: GCD GUI for three numbers

(Summary only)

Modify a GCD program that has a **GUI**, so that it finds the GCD of three numbers.

Section 9

Enabling and disabling components

Aim

AIM: To explore the principle of enabling and disabling **graphical user interface (GUI)** components, and revisit `JButton` and `JTextField`.

Enabling and disabling components

- No example here
 - just some concepts
 - and coursework for you to try them.

- `javax.swing.JButton` has **instance method** `setEnabled()`
 - takes **boolean method parameter**
 - * **false**: button becomes disabled – pressing has no effect
 - * **true**: button becomes enabled.

GUI API: JTextField: setEnabled()

- `javax.swing.JTextField` has **instance method** `setEnabled()`
 - takes `boolean` **method parameter**
 - * `false`: field becomes disabled – cannot change text
 - * `true`: field becomes enabled.

GUI API: JButton: setText ()

- javax.swing.JButton has **instance method** `setText ()`
 - takes `String`
 - changes text label on button.

Coursework: StopClock using a text field and disabled split button

(Summary only)

Modify a stop clock program so that the split time button is disabled when the clock is not running.

Section 10

Example:

Single times table with a GUI

Aim

AIM: To introduce JTextArea and the **layout manager** called BorderLayout.

Single times table with a GUI

- Single times table
 - user enters multiplier in `JTextField`
 - presses `JButton`
 - result shown in `JTextArea`.

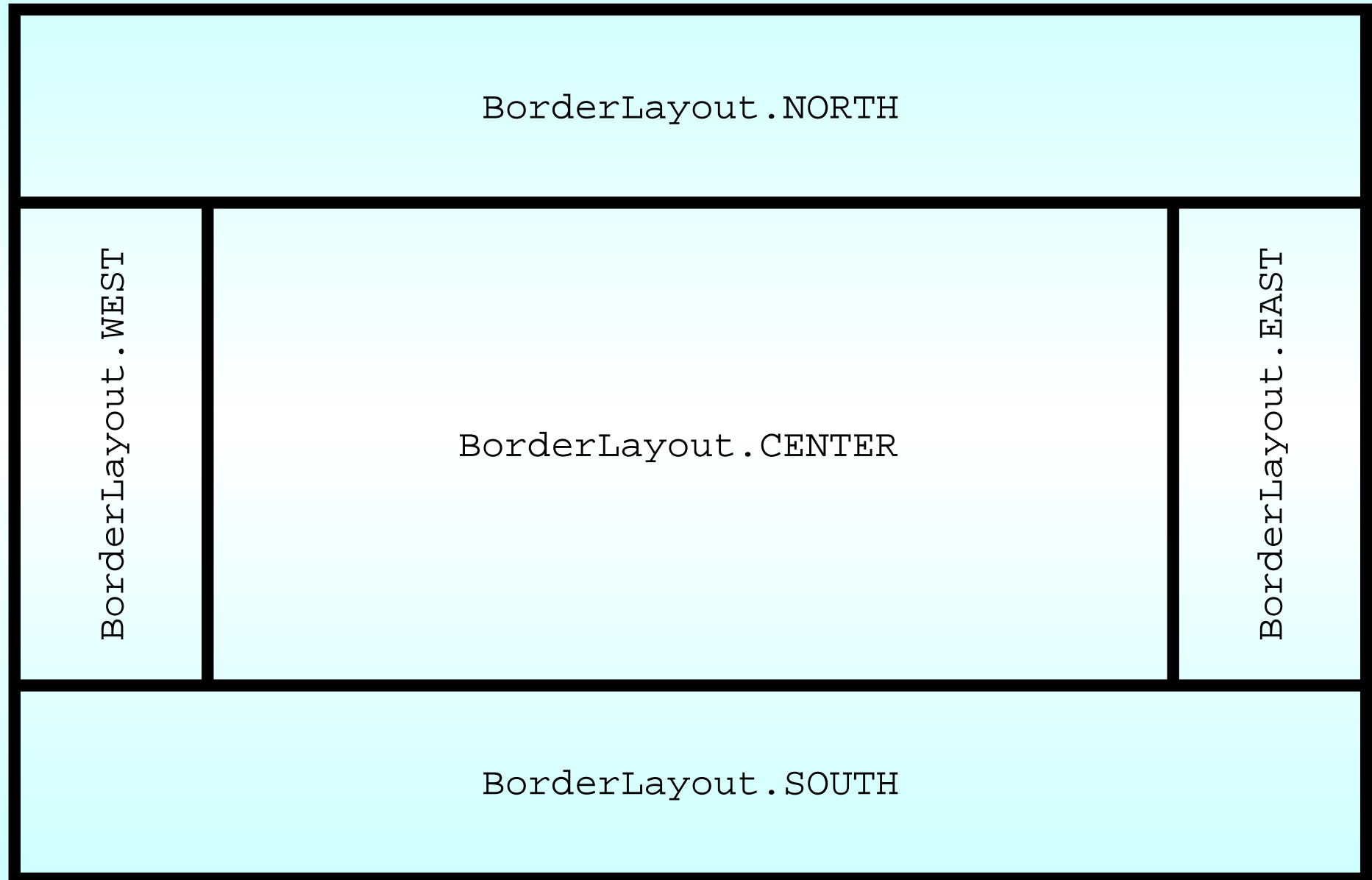
- `javax.swing.JTextArea` part of **GUI**
 - displays larger piece of text
 - multiple lines.
- Size specified as **method arguments constructor method**
 - number of rows (lines)
 - number of columns (characters per line).

Single times table with a GUI

- Shall use a BorderLayout...

- `java.awt.BorderLayout` is **layout manager**
 - slots for five components
 - * centre plus each of four sides.
 - Names of positions modelled as **class constants**
 - * `BorderLayout.CENTER`, `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.WEST`. and `BorderLayout.EAST`.
- Designed for use when one component is main component
 - e.g. `JTextArea` for results
 - placed in centre.
- Others, e.g. buttons, above, below, left and/or right.
- Diagram...

GUI API: LayoutManager: BorderLayout



Single times table with a GUI

- Use only 3 positions
 - multiplier at `BorderLayout.NORTH`
 - result at `BorderLayout.CENTER`
 - button at `BorderLayout.SOUTH`.

```
001: import java.awt.BorderLayout;
```

```
002: import java.awt.Container;
```

```
003: import java.awt.event.ActionEvent;
```

```
004: import java.awt.event.ActionListener;
```

```
005: import javax.swing.JButton;
```

```
006: import javax.swing.JFrame;
```

```
007: import javax.swing.JTextArea;
```

```
008: import javax.swing.JTextField;
```

```
009:
```

Single times table with a GUI

```
010: // Program to show a times table for a multiplier chosen by the user.
011: public class TimesTable extends JFrame implements ActionListener
012: {
013:     // A text field for the user to enter the multiplier.
014:     private final JTextField multiplierJTextField = new JTextField(5);
015:
016:     // A text area for the resulting times table, 15 lines of 20 characters.
017:     private final JTextArea displayJTextArea = new JTextArea(15, 20);
```

- When add components
 - specify position for each one...

GUI API: Container: add(): adding with a position constraint



- `java.awt.Container` has another **instance method** `add()`
 - takes **GUI** component
 - and some other **object** constraining how should be positioned.
- Intended for use with **layout managers** with position constraints
 - e.g. `java.awt.BorderLayout`.
- E.g. following makes `JLabel` appear in north.

```
myContainer.setLayout(new BorderLayout());
```

```
myContainer.add(new JLabel("This is in the north"), BorderLayout.NORTH);
```

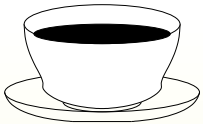
Single times table with a GUI

```
020: // Constructor.
021: public TimesTable()
022: {
023:     setTitle("Times Table");
024:
025:     Container contents = getContentPane();
026:     contents.setLayout(new BorderLayout());
027:
028:     contents.add(multiplierJTextField, BorderLayout.NORTH);
029:     contents.add(displayJTextArea, BorderLayout.CENTER);
030:
031:     JButton displayJButton = new JButton("Display");
032:     contents.add(displayJButton, BorderLayout.SOUTH);
033:     displayJButton.addActionListener(this);
034:
035:     setDefaultCloseOperation(EXIT_ON_CLOSE);
036:     pack();
037: } // TimesTable
```


Single times table with a GUI



Coffee If we wanted to stop the user from typing directly into the
time: text area, what do you imagine we would need to do?



Coffee Before continuing, make a sketch of what you think the
time: TimesTable GUI will look like.

- `actionPerformed()` generates result text
 - places in `JTextArea...`

GUI API: JTextArea: setText ()

- `javax.swing.JTextArea` has **instance method** `setText ()`
 - takes `String` **method argument**
 - changes text to it
 - may contain **new line characters** – separate lines.

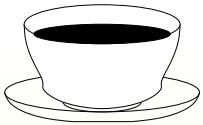
GUI API: JTextArea: append ()

- `javax.swing.JTextArea` has **instance method** `append ()`
 - takes `String`
 - appends onto end of existing text
 - no automatic line breaks.

Single times table with a GUI

```
040: // Act upon the button being pressed.
041: public void actionPerformed(ActionEvent event)
042: {
043:     // Empty the text area to remove any previous result.
044:     displayJTextArea.setText("");
045:
046:     int multiplier = Integer.parseInt(multiplierJTextField.getText());
047:
048:     displayJTextArea.append("-----\n");
049:     displayJTextArea.append("| Times table for " + multiplier + "\n");
050:     displayJTextArea.append("-----\n");
051:     for (int thisNumber = 1; thisNumber <= 10; thisNumber++)
052:         displayJTextArea.append("| " + thisNumber + " x " + multiplier
053:             + " = " + thisNumber * multiplier + "\n");
054:     displayJTextArea.append("-----\n");
055: } // actionPerformed
```

Single times table with a GUI



*Coffee
time:*

What would be the consequence if we missed out the bit that sets the text to empty, given that JTextArea **objects** start with no text anyway?

Single times table with a GUI

- In **main method**
 - create **instance** of TimesTable
 - make it visible.
- Do not need to store **reference** in a **variable!**

```
058: // Create a TimesTable and make it appear on the screen.
059: public static void main(String[] args)
060: {
061:     new TimesTable().setVisible(true);
062: } // main
063:
064: } // class TimesTable
```

Trying it

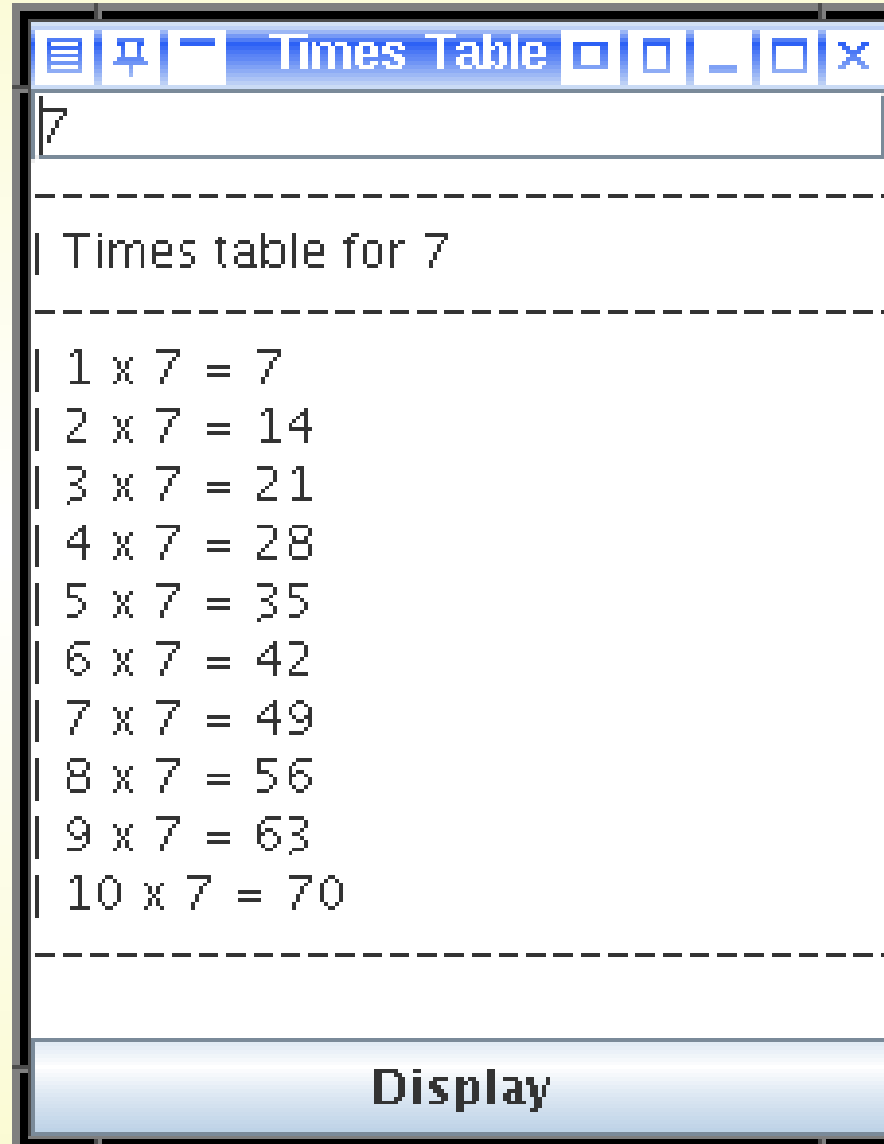
Console Input / Output

```
$ java TimesTable
```

```
$ _
```

Run

Trying it



Trying it

*Coffee
time:*

Look back at the main methods of the previous examples in this chapter. For which ones could we have decided *not* to store the reference to the instance of the GUI in a variable? Given that these variables are **method variables** and so are 'destroyed' when the main method ends, does it actually make any difference to the Java **virtual machine** whether or not we use them to temporarily store a reference to the GUI?



Section 11

Example: GCD with Panels

Aim

AIM: To introduce the idea of using `JPanel` **objects** to make a more sophisticated interface.

GCD with Panels

- Reimplement GCD program – nicer interface.
- Use `JPanel` to enable more layout control...

- `javax.swing.JPanel`
 - **extension** of (older) `java.awt.Container`
 - thus contains other components
 - has `add()` **instance methods**.
- Designed to work well with **Java Swing**
 - recommended kind of container for grouping collection of components to be treated as one for layout.

- Improved **GUI** plan:
 - GridLayout to obtain two rows, one column i.e two by one
 - each of these will be JPanel
 - top one will use GridLayout, two by two
 - bottom one will use GridLayout, one by two
 - right one of this will be JPanel with two by one GridLayout!
- Diagram...

GCD with Panels

ContentPane with GridLayout, 2 x 1

JPanel with GridLayout, 2 x 2

JLabel "Number 1"

JLabel "Number 2"

JTextField for number1

JTextField for number2

JPanel with GridLayout, 1 x 2

JButton labelled "Compute"

JPanel with GridLayout, 2 x 1

JLabel "GCD of No 1 and No 2"

JTextField for result

GCD with Panels

```
001: import java.awt.Container;
002: import java.awt.GridLayout;
003: import java.awt.event.ActionEvent;
004: import java.awt.event.ActionListener;
005: import javax.swing.JButton;
006: import javax.swing.JFrame;
007: import javax.swing.JLabel;
008: import javax.swing.JPanel;
009: import javax.swing.JTextField;
010:
011: // Calculates the GCD of two integers.
012: public class GCD extends JFrame implements ActionListener
013: {
014:     // A JTextField for each number.
015:     private final JTextField number1JTextField = new JTextField(20);
016:     private final JTextField number2JTextField = new JTextField(20);
017:
018:     // A JTextField for the result.
019:     private final JTextField resultJTextField = new JTextField(20);
```

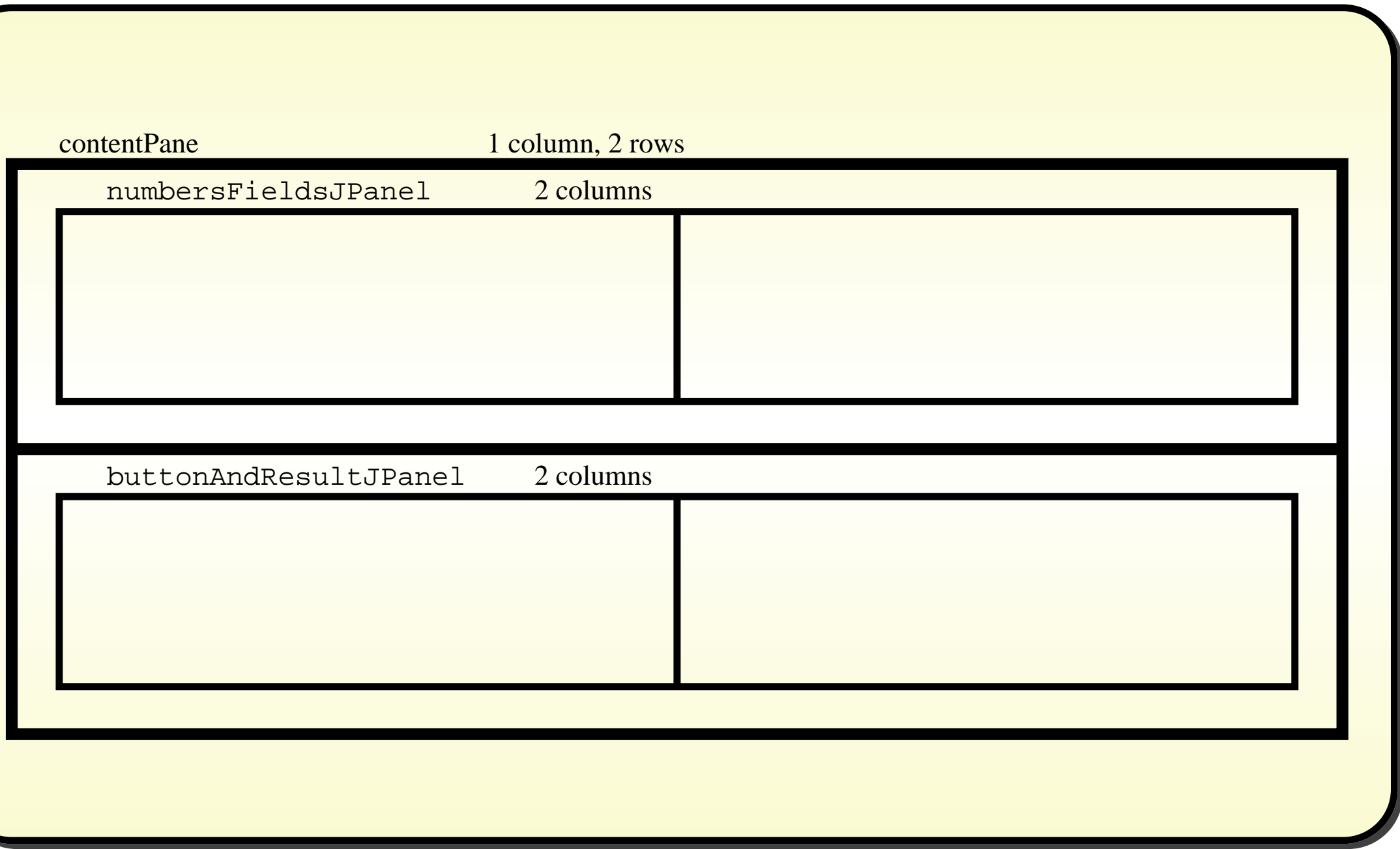

GCD with Panels

```
022: // Constructor.
023: public GCD()
024: {
025:     setTitle("GCD");
026:
027:     Container contents = getContentPane();
028:     // Main layout will be 2 by 1.
029:     contents.setLayout(new GridLayout(0, 1));
030:
031:     // A JPanel for the top half of the main grid.
032:     // This will have a layout of 2 by 2.
033:     // It will contain two labels, and two text fields for input.
034:     JPanel numberFieldsJPanel = new JPanel();
035:     contents.add(numberFieldsJPanel);
036:     numberFieldsJPanel.setLayout(new GridLayout(0, 2));
037:
```

```
038: // A JPanel for the bottom half of the main grid.
039: // This will have a layout of 1 by 2.
040: // It will contain the button and JPanel for the result.
041: JPanel buttonAndResultJPanel = new JPanel();
042: contents.add(buttonAndResultJPanel);
043: buttonAndResultJPanel.setLayout(new GridLayout(0, 2));
```

- So far...

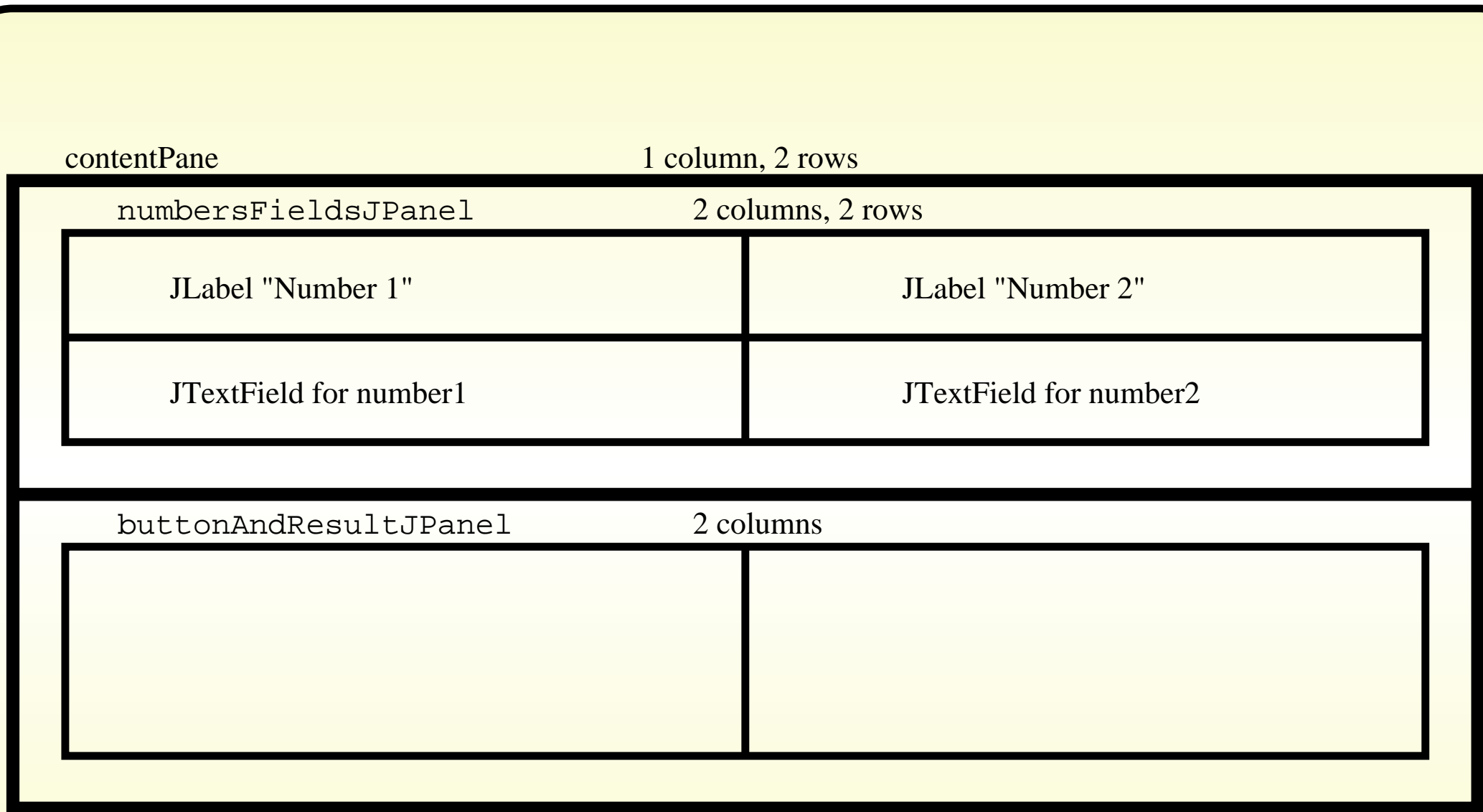
GCD with Panels



```
045: // Two labels and two text fields for the top JPanel.  
046: numberFieldsJPanel.add(new JLabel("Number 1"));  
047: numberFieldsJPanel.add(new JLabel("Number 2"));  
048: numberFieldsJPanel.add(number1JTextField);  
049: numberFieldsJPanel.add(number2JTextField);
```

- The above four components in two columns will use two rows...

GCD with Panels

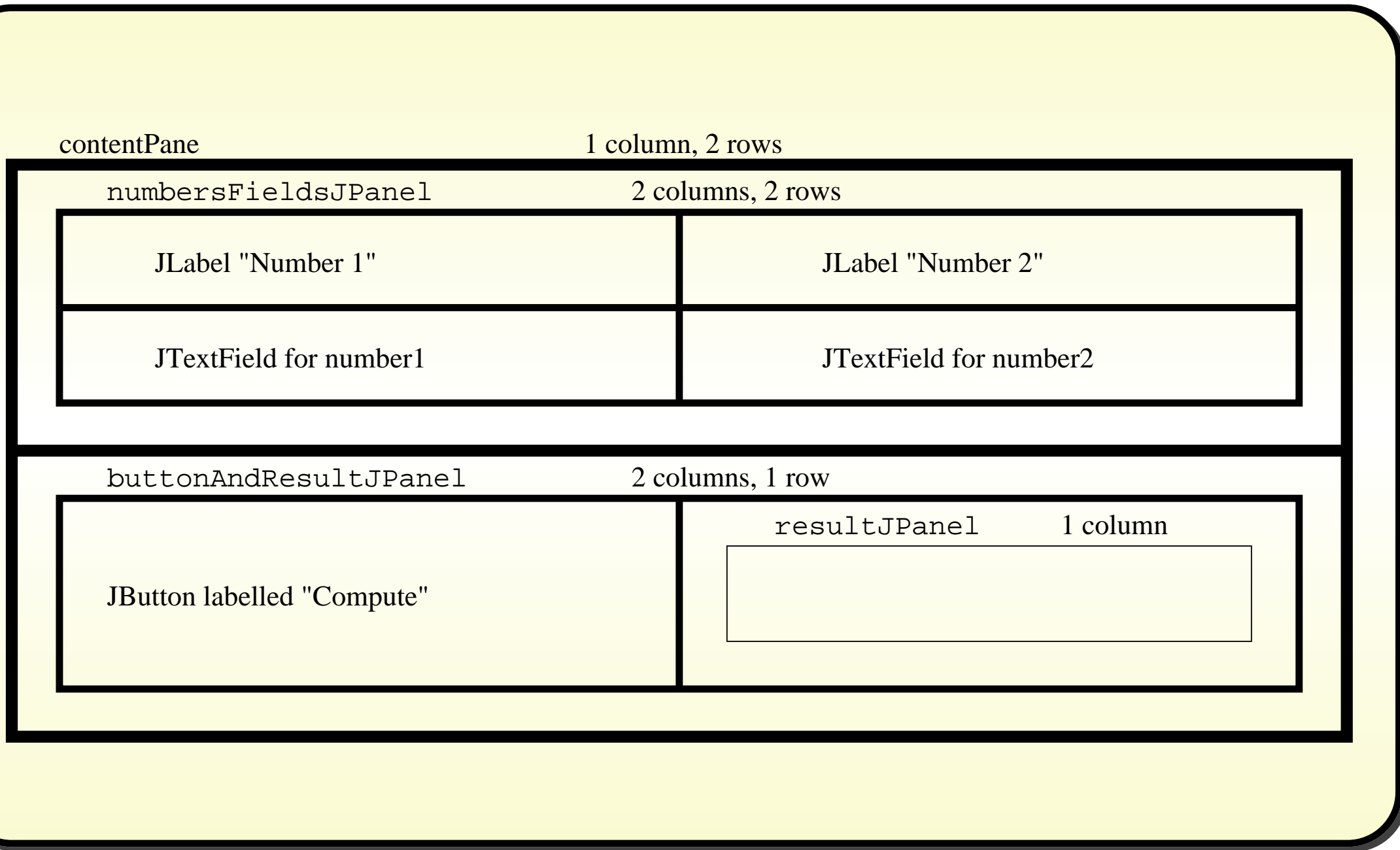


GCD with Panels

```
051: // The compute button will live in the left of the bottom JPanel.
052: JButton computeJButton = new JButton("Compute");
053: buttonAndResultJPanel.add(computeJButton);
054: computeJButton.addActionListener(this);
055:
056: // A JPanel for the right of the bottom half of the main grid.
057: // This will have a layout of 2 by 1.
058: // It will contain a label and a text field for the result.
059: JPanel resultJPanel = new JPanel();
060: buttonAndResultJPanel.add(resultJPanel);
061: resultJPanel.setLayout(new GridLayout(0, 1));
```

- So far...

GCD with Panels



GCD with Panels

- Finally:

```
063:    // A label and a text field for the bottom right JPanel.  
064:    resultJPanel.add(new JLabel("GCD of Number 1 and Number 2"));  
065:    resultJPanel.add(resultJTextField);  
  
066:  
067:    setDefaultCloseOperation(EXIT_ON_CLOSE);  
068:    pack();  
069: } // GCD
```



Coffee time: Convince yourself that the above code will result in a GUI that is laid out as we planned.

GCD with Panels

```
072: // Act upon the button being pressed.
073: public void actionPerformed(ActionEvent event)
074: {
075:     int number1 = Integer.parseInt(number1JTextField.getText());
076:     int number2 = Integer.parseInt(number2JTextField.getText());
077:     int theGCD = MyMath.greatestCommonDivisor(number1, number2);
078:     resultJTextField.setText("" + theGCD);
079: } // actionPerformed
080:
081:
082: // Create a GCD and make it appear on screen.
083: public static void main(String[] args)
084: {
085:     new GCD().setVisible(true);
086: } // main
087:
088: } // class GCD
```

Trying it

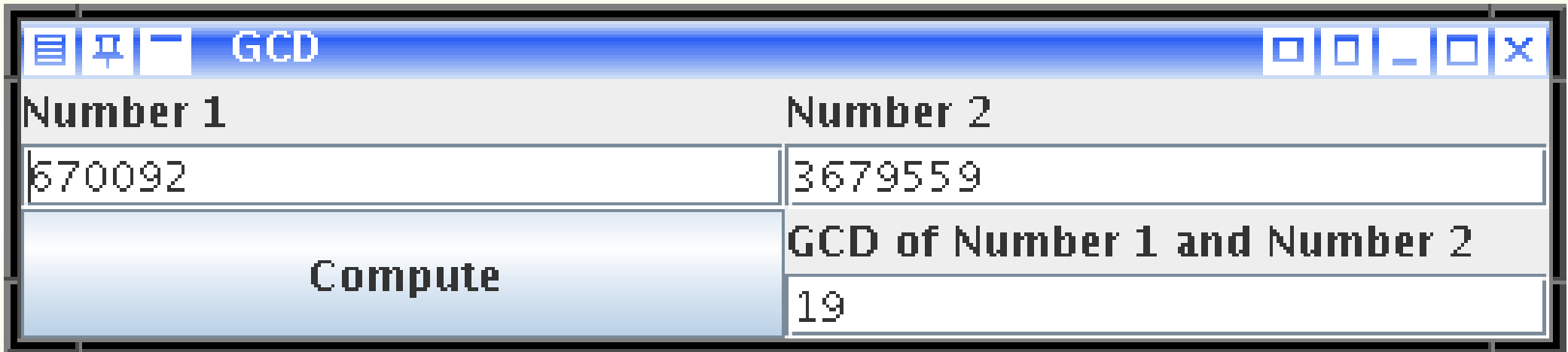
Console Input / Output

```
$ java GCD
```

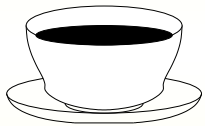
```
$ _
```

Run

Trying it



Trying it



*Coffee
time:*

Can you think of other ways that we could have obtained the same layout?

Section 12

Example:

Single times table with a
ScrollPane

Aim

AIM: To introduce the use of JScrollPane and revisit
JTextField.

Single times table with a JScrollPane

- Revisit TimesTable
 - add second `JTextField` to let user choose how many rows.
- Might have more rows than space allocated for result
 - so make text scrollable.
- `JTextArea` does not provide scrolling itself
 - instead use `JScrollPane`.

- A `javax.swing.JScrollPane` object
 - provides scrolling facility for another **GUI** component.
- Simplest use: provide other component as argument to constructor.
- E.g. Add `JTextArea` to **content pane** of `JFrame`
 - without scrolling.

```
Container contents = getContentPane();  
contents.add(new JTextArea(15, 20));
```

- Now with scrolling.

```
Container contents = getContentPane();  
contents.add(new JScrollPane(new JTextArea(15, 20)));
```


Single times table with a JScrollPane

```
001: import java.awt.BorderLayout;
002: import java.awt.Container;
003: import java.awt.GridLayout;
004: import java.awt.event.ActionEvent;
005: import java.awt.event.ActionListener;
006: import javax.swing.JButton;
007: import javax.swing.JFrame;
008: import javax.swing.JLabel;
009: import javax.swing.JPanel;
010: import javax.swing.JScrollPane;
011: import javax.swing.JTextArea;
012: import javax.swing.JTextField;
013:
014: // Program to show a times table for a multiplier chosen by the user.
015: // The user also chooses the size of the table.
016: public class TimesTable extends JFrame implements ActionListener
017: {
```

- `javax.swing.JTextField` has another **constructor method**
 - takes `String`
 - sets as initial value.

```
JTextField nameJTextField = new JTextField("Type your name here.");
```

Single times table with a ScrollPane

```
018: // A text field for the user to enter the multiplier.
019: private final JTextField multiplierJTextField = new JTextField(5);
020:
021: // A text field for the user to enter the table size, initial value 10.
022: private final JTextField tableSizeJTextField = new JTextField("10");
023:
024: // A text area for the resulting times table, 15 lines of 20 characters.
025: private final JTextArea displayJTextArea = new JTextArea(15, 20);
```

Single times table with a ScrollPane

```
028: // Constructor.
029: public TimesTable()
030: {
031:     setTitle("Times Table");
032:
033:     Container contents = getContentPane();
034:     contents.setLayout(new BorderLayout());
035:
036:     // A JPanel for the two text fields.
037:     // It will be a GridLayout of two times two,
038:     // at the top of the JFrame contents.
039:     JPanel numbersPanel = new JPanel();
040:     contents.add(numbersPanel, BorderLayout.NORTH);
041:     numbersPanel.setLayout(new GridLayout(2, 0));
042:
043:     // Add two JLabels, and two JTextFields to the numbersPanel.
044:     numbersPanel.add(new JLabel("Multiplier:"));
045:     numbersPanel.add(multiplierJTextField);
046:     numbersPanel.add(new JLabel("Table size:"));
047:     numbersPanel.add(tableSizeJTextField);
```

Single times table with a JScrollPane

```
048:
049:     // The result JScrollPane/JTextArea goes in the centre.
050:     contents.add(new JScrollPane(displayJTextArea), BorderLayout.CENTER);
051:
052:     // The JButton goes at the bottom.
053:     JButton displayJButton = new JButton("Display");
054:     contents.add(displayJButton, BorderLayout.SOUTH);
055:     displayJButton.addActionListener(this);
056:
057:     setDefaultCloseOperation(EXIT_ON_CLOSE);
058:     pack();
059: } // TimesTable
```

Single times table with a ScrollPane

```
062: // Act upon the button being pressed.
063: public void actionPerformed(ActionEvent event)
064: {
065:     // Empty the text area to remove any previous result.
066:     displayJTextArea.setText("");
067:
068:     int multiplier = Integer.parseInt(multiplierJTextField.getText());
069:     int tableSize = Integer.parseInt(tableSizeJTextField.getText());
070:
071:     displayJTextArea.append("-----\n");
072:     displayJTextArea.append("| Times table for " + multiplier + "\n");
073:     displayJTextArea.append("-----\n");
074:     for (int thisNumber = 1; thisNumber <= tableSize; thisNumber++)
075:         displayJTextArea.append("| " + thisNumber + " x " + multiplier
076:             + " = " + thisNumber * multiplier + "\n");
077:     displayJTextArea.append("-----\n");
078: } // actionPerformed
```

Single times table with a ScrollPane

```
081: // Create a TimesTable and make it appear on the screen.
082: public static void main(String[] args)
083: {
084:     new TimesTable().setVisible(true);
085: } // main
086:
087: } // class TimesTable
```

Trying it

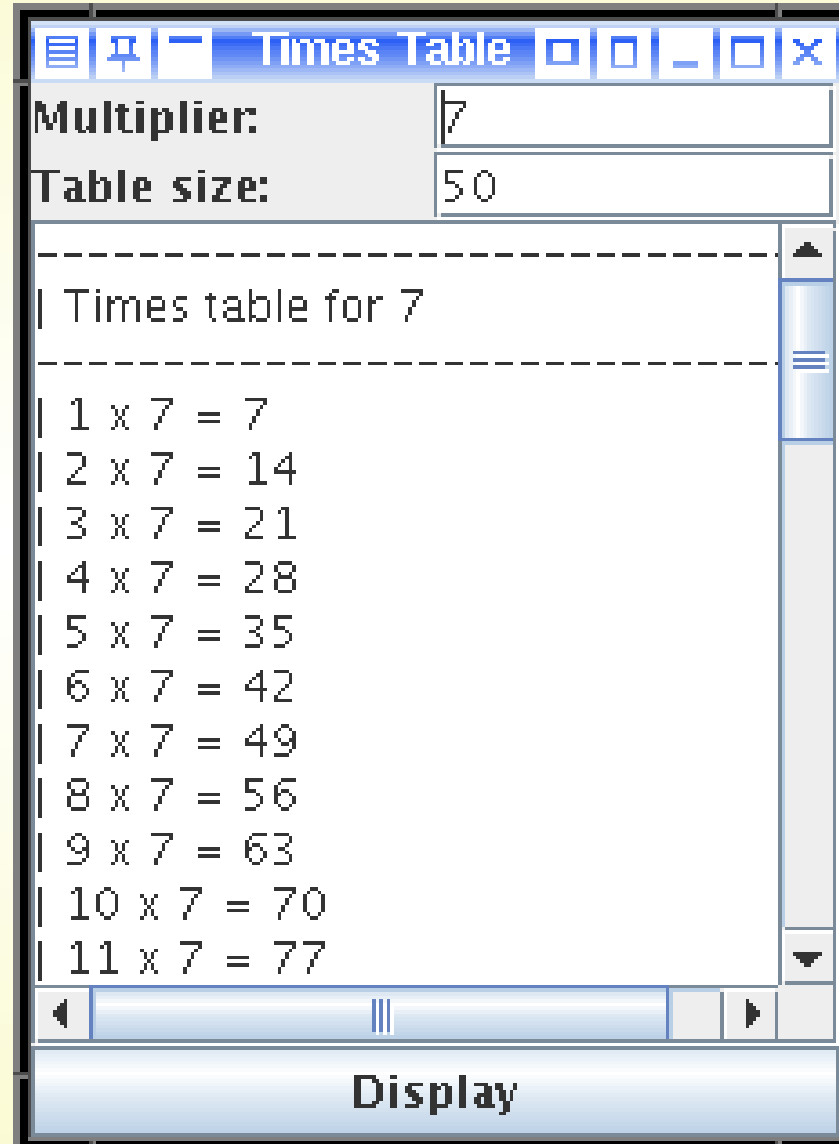
Console Input / Output

```
$ java TimesTable  
$_
```

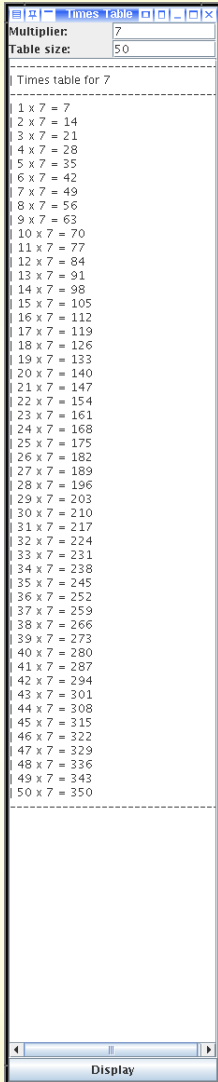
Run

- Window will adjust layout when we change its size
 - beauty of **layout managers**.

Trying it



Trying it



(Summary only)

Write a **GUI** version of the program to show the weights that are obtainable on a balance scale using three weights.

Section 13

Example:

Age history with a GUI

Aim

AIM: To reinforce the **graphical user interface (GUI)** concepts with an example having two `JButtons` and many `JFrames`, for which we revisit `FlowLayout` and `ActionEvent`.

Age history with a GUI

- **GUI** version of AgeHistory
 - reuse Person from previous version
 - and our improved reusable Date
 - just need GUI version of main class.
- Set current date, name and birthday in text fields
- Obtain age history in text area.
- For next person either
 - replace details and obtain again
 - or obtain separate copy of window
 - * any number we wish!

Age history with a GUI

```
001: import java.awt.BorderLayout;
002: import java.awt.Container;
003: import java.awt.FlowLayout;
004: import java.awt.GridLayout;
005: import java.awt.event.ActionEvent;
006: import java.awt.event.ActionListener;
007: import javax.swing.JButton;
008: import javax.swing.JFrame;
009: import javax.swing.JLabel;
010: import javax.swing.JPanel;
011: import javax.swing.JScrollPane;
012: import javax.swing.JTextArea;
013: import javax.swing.JTextField;
014:
015: /* Report the age history of a person.
016:    Current date and person details are entered through text fields.
017:    The result is displayed in a text area.
018:    A ``new'' button enables multiple displays.
019: */
020: public class AgeHistory extends JFrame implements ActionListener
021: {
```

Age history with a GUI

```
022: // JTextFields for the present date.
023: private final JTextField presentDayJTextField = new JTextField(2);
024: private final JTextField presentMonthJTextField = new JTextField(2);
025: private final JTextField presentYearJTextField = new JTextField(4);
026:
027: // JTextFields for the name and birthday.
028: private final JTextField nameJTextField = new JTextField(15);
029: private final JTextField birthDayJTextField = new JTextField(2);
030: private final JTextField birthMonthJTextField = new JTextField(2);
031: private final JTextField birthYearJTextField = new JTextField(4);
032:
033: // JTextArea for the result.
034: private final JTextArea ageHistoryJTextArea = new JTextArea(15, 20);
```


Age history with a GUI

```
036: // The age history display button.  
037: private final JButton displayJButton = new JButton("Display");  
038:  
039: // The new window button.  
040: private final JButton newJButton = new JButton("New");
```

Age history with a GUI

```
042: // The number of instances created: each has its number in the title.
043: private static int instanceCountSoFar = 0;
044:
045:
046: // Constructor.
047: public AgeHistory()
048: {
049:     instanceCountSoFar++;
050:     setTitle("Age History (" + instanceCountSoFar + ")");
```



Coffee time: What would happen if we omitted the **reserved word** `static` from the declaration of `instanceCountSoFar`?

Age history with a GUI

```
052:     Container contents = getContentPane();
053:     contents.setLayout(new BorderLayout());
054:
055:     // The top panel is for the inputs.
056:     // It will be a grid of 3 by 2.
057:     JPanel inputDataJPanel = new JPanel();
058:     contents.add(inputDataJPanel, BorderLayout.NORTH);
059:     inputDataJPanel.setLayout(new GridLayout(0, 2));
060:
061:     // Top left of inputDataJPanel.
062:     inputDataJPanel.add(new JLabel("Present date"));
```

- Next add another JPanel
 - with FlowLayout having a left alignment.

- `java.awt.FlowLayout` can be given alignment mode
 - as **method argument** to one **constructor method**.
- Affects behaviour when component is larger than minimum needed for its contents.
- An `int` value
 - * `FlowLayout.CENTER` – the laid out items are centred in the container.
 - * `FlowLayout.LEFT` – the laid out items are on the left of the container, with unused space on the right.
 - * `FlowLayout.RIGHT` – the laid out items are on the right of the container, with unused space on the left.
 - Default – centre.

Age history with a GUI

```
064:    // Top right of inputDataJPanel.
065:    // A JPanel with left aligned FlowLayout,
066:    // For today's date components.
067:    JPanel presentDayJPanel = new JPanel();
068:    inputDataJPanel.add(presentDayJPanel);
069:    presentDayJPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
070:
071:    // JTextFields for present date components, with JLabels.
072:    presentDayJPanel.add(presentDayJTextField);
073:    presentDayJPanel.add(new JLabel("/"));
074:    presentDayJPanel.add(presentMonthJTextField);
075:    presentDayJPanel.add(new JLabel("/"));
076:    presentDayJPanel.add(presentYearJTextField);
```

Age history with a GUI

```
078:    // Middle left of inputDataJPanel.
079:    inputDataJPanel.add(new JLabel("Person name"));
080:
081:    // Middle right of inputDataJPanel.
082:    // Use a JPanel so that alignment matches rows above and below.
083:    JPanel nameJPanel = new JPanel();
084:    inputDataJPanel.add(nameJPanel);
085:    nameJPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
086:    nameJPanel.add(nameJTextField);
087:
088:    // Bottom left of inputDataJPanel.
089:    inputDataJPanel.add(new JLabel("Birthday"));
```

Age history with a GUI

```
091:    // Bottom right of inputDataJPanel.
092:    // A JPanel with left aligned FlowLayout,
093:    // For birthday components.
094:    JPanel birthdayJPanel = new JPanel();
095:    inputDataJPanel.add(birthdayJPanel);
096:    birthdayJPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
097:
098:    // JTextFields for birthday components, with JLabels.
099:    birthdayJPanel.add(birthDayJTextField);
100:    birthdayJPanel.add(new JLabel("/"));
101:    birthdayJPanel.add(birthMonthJTextField);
102:    birthdayJPanel.add(new JLabel("/"));
103:    birthdayJPanel.add(birthYearJTextField);
```

Age history with a GUI

```
105:    // The result JTextArea goes in the centre.
106:    contents.add(new JScrollPane(ageHistoryJTextArea), BorderLayout.CENTER);
107:
108:    // The buttons go at the bottom, in a JPanel with a FlowLayout.
109:    JPanel buttonJPanel = new JPanel();
110:    contents.add(buttonJPanel, BorderLayout.SOUTH);
111:    buttonJPanel.setLayout(new FlowLayout());
112:    buttonJPanel.add(displayJButton);
113:    displayJButton.addActionListener(this);
114:    buttonJPanel.add(newJButton);
115:    newJButton.addActionListener(this);
```


Age history with a GUI

- The present date may have been set in a previous window.

```
117:    // Allow for the possibility that the present date has already been set.
118:    Date presentDate = Date.getPresentDate();
119:    if (presentDate != null)
120:    {
121:        presentDayJTextField.setText("" + presentDate.getDay());
122:        presentMonthJTextField.setText("" + presentDate.getMonth());
123:        presentYearJTextField.setText("" + presentDate.getYear());
124:        presentDayJTextField.setEnabled(false);
125:        presentMonthJTextField.setEnabled(false);
126:        presentYearJTextField.setEnabled(false);
127:    } // if
```

Age history with a GUI

- Closing window should *not* end the program
 - program will automatically exit once last window has been disposed.

```
129:     setDefaultCloseOperation(DISPOSE_ON_CLOSE);  
130:     pack();  
131: } // AgeHistory
```



Coffee time: Before continuing, make a sketch of what you think the AgeHistory GUI will look like.

- Our `actionPerformed()` must determine which button was pressed...

GUI API: `ActionEvent: getSource()`

- `java.awt.event.ActionEvent` has **instance method** `getSource()`
 - **returns reference** to **object** that caused **event**.

Age history with a GUI

```
134: // Act upon the button being pressed.
135: public void actionPerformed(ActionEvent event)
136: {
137:     if (event.getSource() == newJButton)
138:         new AgeHistory().setVisible(true);
139:
140:     else if (event.getSource() == displayJButton)
141:     {
142:         // Set the present date only if it has not already been set.
143:         if (Date.getPresentDate() == null)
144:         {
145:             Date presentDay
146:                 = new Date(Integer.parseInt(presentDayJTextField.getText()),
147:                             Integer.parseInt(presentMonthJTextField.getText()),
148:                             Integer.parseInt(presentYearJTextField.getText()))
149:                 );
150:             Date.setPresentDate(presentDay);
```

Age history with a GUI

```
151:         // Date should be set only once: disable further date setting.
152:         presentDayJTextField.setEnabled(false);
153:         presentMonthJTextField.setEnabled(false);
154:         presentYearJTextField.setEnabled(false);
155:     } // if
156:     // Compute and display the age history.
157:     String name = nameJTextField.getText();
158:     Date birthday
159:         = new Date(Integer.parseInt(birthDayJTextField.getText()),
160:                 Integer.parseInt(birthMonthJTextField.getText()),
161:                 Integer.parseInt(birthYearJTextField.getText()));
162:     };
163:     Person person = new Person(name, birthday);
164:     ageHistoryJTextArea.setText(person.ageHistory());
165: } // else if
166: } // actionPerformed
```

Age history with a GUI

```
169: // Create an AgeHistory and make it appear on screen.
170: public static void main(String[] args)
171: {
172:     // Ensure we use just \n for age history line separator on all platforms.
173:     Person.setLineSeparator("\n");
174:     new AgeHistory().setVisible(true);
175: } // main
176:
177: } // class AgeHistory
```

Trying it

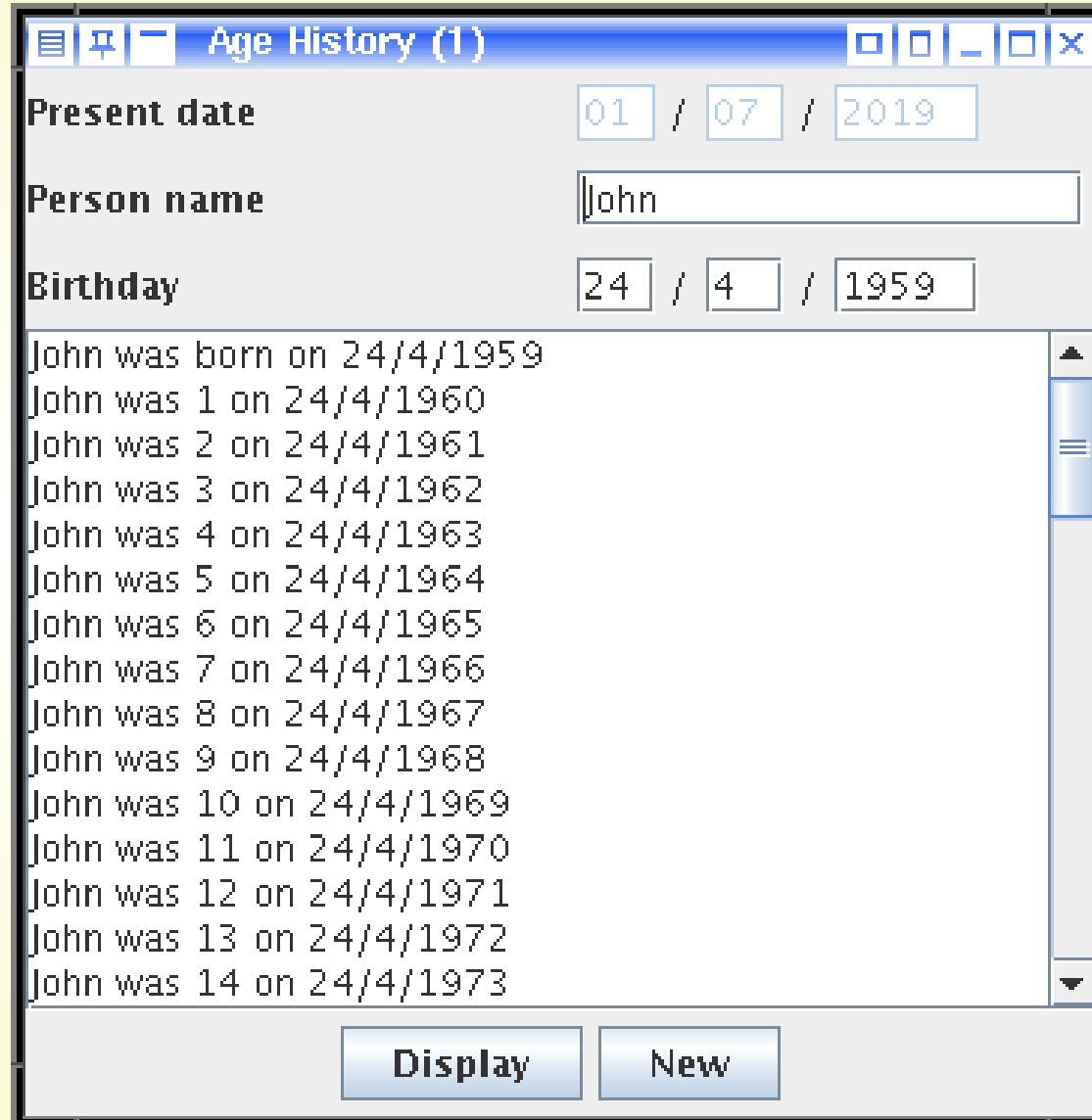
Console Input / Output

```
$ java AgeHistory  
$_
```

Run

Create more windows...

Trying it



Age History (1)

Present date 01 / 07 / 2019

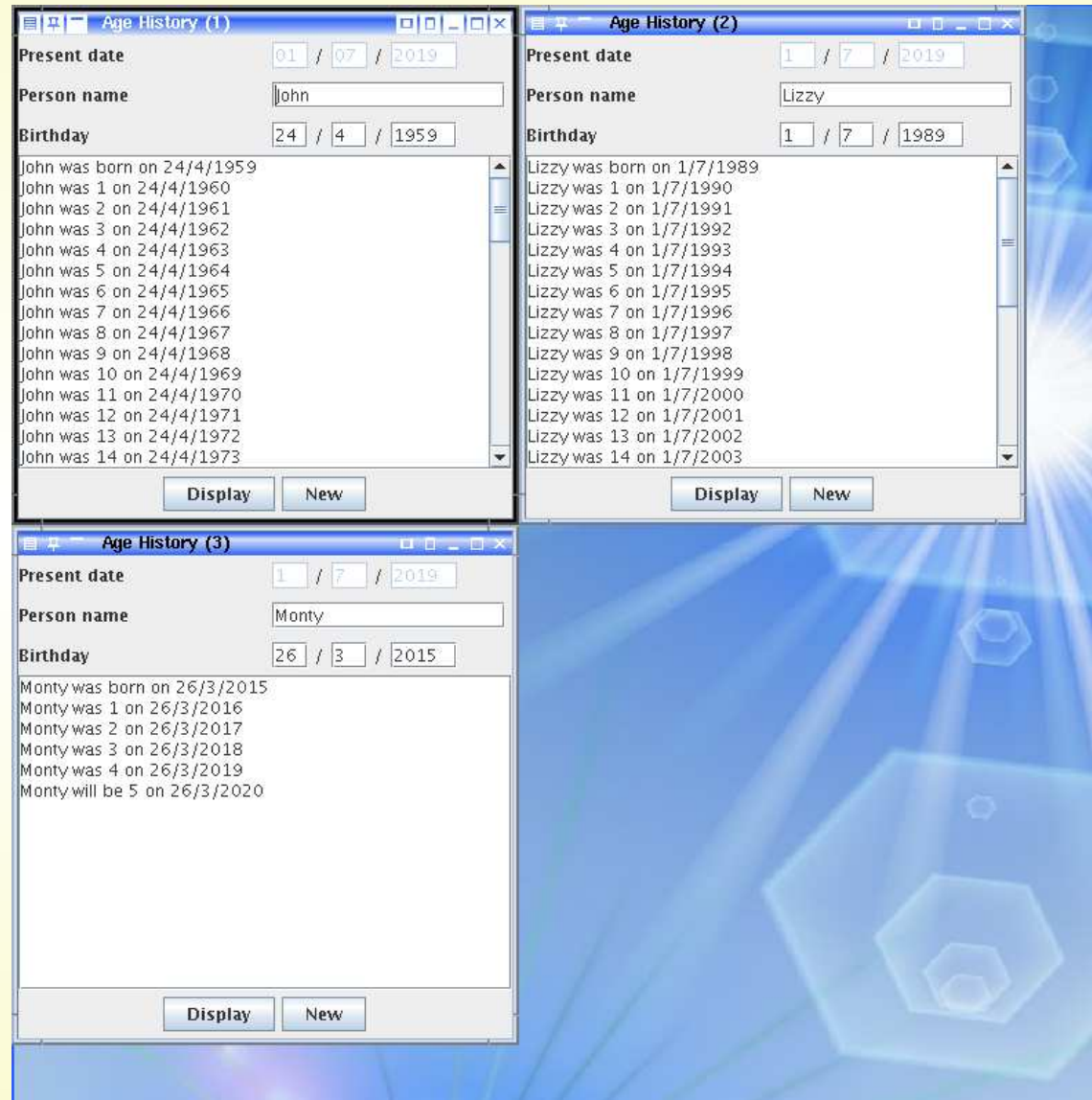
Person name John

Birthday 24 / 4 / 1959

John was born on 24/4/1959
John was 1 on 24/4/1960
John was 2 on 24/4/1961
John was 3 on 24/4/1962
John was 4 on 24/4/1963
John was 5 on 24/4/1964
John was 6 on 24/4/1965
John was 7 on 24/4/1966
John was 8 on 24/4/1967
John was 9 on 24/4/1968
John was 10 on 24/4/1969
John was 11 on 24/4/1970
John was 12 on 24/4/1971
John was 13 on 24/4/1972
John was 14 on 24/4/1973

Display New

Trying it



Trying it

*Coffee
time:*

Our example here has a subtle **design bug**. Figure out how we can end up with a date being shown in the text fields for the present date which is not the value that is actually being used for the present date! (Hint: what if the `New` button is pressed before the `Display` button?) What is a simple fix for this problem?



Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
 - a self-test question,
 - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.