## New architecture and painting properly

- In this Section:
  - A better structure for our Java applications
  - ... and an interesting consequence
  - Improving drawn graphics – "painting properly"





## Java application architecture

- Java classes can be used to build applications in many different ways
- The only specific requirement is that the main class used to launch the application has

```
public static void main(String [] args) { ... }
```

- We have been using the JFS application architecture
  - Quite simple
  - Main program is *all in one class*
  - But can look rather confusing
  - Reminder: See next slide...

```
public class Example extends JFrame
                     implements ChangeListener {
   private ...;
   public static void main (String[] args) {
      Example frame = new Example();
      frame.setSize(400,100);
      frame.setLocation(50,50);
      frame.setTitle("Title");
      frame.createGUI();
      frame.setVisible(true);
   }
   private void createGUI() { ... }
   public void stateChanged(ChangeEvent e) { ... }
   ...
}
```

Possible confusion





- For a clearer arrangement, we will separate:
  - The main launcher component – one class
  - The GUI component – another class
  - (and there may be other supporting classes too)
- The main launcher looks like this:

```
public class Example {
   public static void main (String[] args) {
      ExampleGUI frame = new ExampleGUI("Title");
      frame.setLocation(50,50);
      frame.setVisible(true);
   }
}
```

Note: Constructor

- This organizes *instantiation* and *deployment* of the GUI
  - But not its internal details

- And the GUI class looks like this:

```
public class ExampleGUI extends JFrame
                        implements ChangeListener {
   private ...;
   public ExampleGUI(String title) {
      setSize(400,100);
      setTitle(title);
      createGUI();
   }
   private void createGUI() { ... }
   public void stateChanged(ChangeEvent e) { ... }
   ...
}
```

Note: Constructor

Note: No `frame.` as the methods are in *this* class

- This builds the window but does not control the when/where





- So, the launch process is:
  - Indicate the main class to the JVM
  - JVM finds the `main` method in it, calls it
  - `main` creates an instance of GUI class
  - GUI constructor is called, builds window
  - `main` places window and makes it visible
  - Event handling starts

## An interesting consequence

- Remember:
  - `new ExampleGUI(...)` *builds a complete new instance*
- We could have *as many* `StudentRecord`s *as we wished*
  - So too with `ExampleGUI`s!
- So, the launcher `main` method could look like this:

```
public static void main (String[] args) {
  ExampleGUI frame1 = new ExampleGUI("Title 1");
  frame1.setLocation(50,50);
  frame1.setVisible(true);
  ExampleGUI frame2 = new ExampleGUI("Title 2");
  frame2.setLocation(100,100);
  frame2.setVisible(true);
}
```

- There will be two independently operating windows!





## Painting properly (not in JFS)

- This is a secret that Bell & Parr chose to hide in JFS 4th edition onwards!
  - They claim that it is hard to understand (I don't agree)
- What is the problem?
  - Your drawings *do not automatically refresh* when the window is created/restored/exposed
  - You may already have been irritated by this!
  - There is no problem with applications that use only text fields, buttons, sliders, etc
- Example: The QuickBlind application from the lecture on Decisions and Buttons:

## Analysis

- There seems to be some unfair "magic" at work:
  - JButtons, JSliders, JLabels, etc all seem to re-draw themselves automatically when the window needs refreshing
  - If our JPanel has a background colour set (`setBackground`), then it is filled in automatically
- But ***our*** drawing/painting actions are only carried out when an ordinary event occurs, for example a button click
  - How can we make them automatic?
- The "magic" works like this:
  - The OS window manager tells the JVM when a window needs refreshing
  - This causes a `PaintEvent`...
  - ... and the JVM calls *every visible widget's* `paintComponent` event handler method





- Every widget has a method:

```
public void paintComponent(Graphics g)
```

  - The JVM automatically ***clears*** the widget's screen area
  - and passes a suitable `Graphics` for that screen area to the method
  - The method then draws the widget's specific appearance
- Clear enough for JButton, JSlider, etc
- But JPanel's `paintComponent` is *built into the library class* and *cannot know* about our drawing actions!
- So, there is *no* magic! We can exploit this knowledge and play a clever game:
  - We can create an *"extended"* JPanel...
  - ... *"overriding"* its `paintComponent` with one of our design!
  - "Overriding" = "replacing"

## A QuickBlind that paints properly

- Here is one way to organize the program:
- Where we create a new JPanel, we now have:

```
panel = new JPanel() {
   public void paintComponent(Graphics g) {
      super.paintComponent(g); // Important!
      paintScreen(g);          // Our drawing
   }
};
```

- `panel` is assigned a new JPanel object but with its `paintComponent` *overridden by a new method*
- The new method *must* call the superclass version to do its work (filling in the set background colour)…
- …and then call *our own* `paintScreen` method – to do the real drawing





- The extended JPanel:
  - Is called an "anonymous inner class"
  - Its bytecode is placed in a file with a name like `WindowBlind$1.class`
- `paintScreen` is quite straightforward
  - ***It can assume that it is given an erased graphics area***
  - All it needs to do is draw the correct picture, e.g:

```
private void paintScreen(Graphics g) {
   int sliderValue = slider.getValue();
   g.setColor(Color.black);
   g.drawRect(120, 80, 60, 100);
   g.fillRect(120, 80, 60, sliderValue);
}
```

- That deals with window refreshing
  - *But our own event handlers can now exploit this*

- So, finally, we *remove* the calls of `paintScreen` (or explicit drawing code) from the event handlers `stateChanged` and `actionPerformed`
  - Instead we instruct the JVM ***to force a screen refresh*** - a "fictitious" `PaintEvent`:
  - … which will *indirectly* cause our `paintComponent` to be called

```
public void stateChanged(ChangeEvent e) {
    repaint();  // Force a screen refresh
}
public void actionPerformed(ActionEvent e) {
    if (e.getSource()==open)
        ...
    if (e.getSource()==close)
        ...
    repaint();  // Force a screen refresh
}
```





- Notes:
  - `repaint();` does *not* cause an immediate painting process
  - It instructs the JVM to add a `PaintEvent` to its queue of events to be dealt with
  - Then when the current event handler finishes (returns), the `PaintEvent` "happens"
  - (The JVM "coalesces" multiple `PaintEvents`)
- Here is the improved version running:
- Note:
  - The window & blind are drawn immediately
  - And refresh properly on minimize/restore

Demo

**End of section**