Java Interfaces
Part 1 - Events
Version 1.1

By
Dr. Nicholas Duchon
July 22, 2007

Overview

- Philosophy
- Large Scale Java Language Structures
- Abstract classes
  - Declarations
  - Extending a abstract classes
- Interfaces
  - Declarations
  - Interface hierarchy
  - Implementing interfaces
  - Generics
- Using interfaces effectively
  - Why bother with this complexity?

Philosophy – Part 1

- Why did I write these slides and what viewpoint am I taking?
  - I have seen one too many author state that “interfaces are easy” or “interfaces make programming simple”. I am now firmly convinced that for beginning programmers, interfaces are anything but easy, and they do not make simple programs simple.
  - For example – consider a common interface: an electrical plug and socket in your house. The plug itself seems very simple, but in reality, it satisfies all sorts of standards, and if it didn't, it would not work. The two sides of the interface are extremely complicated. The socket side starts with the socket, the house wiring, the connection to the street power grid, then a complicated power grid eventually connects through transformers and switches to generators.
  - On the other side of the plug could be something as simple as a light, or as complicated as a computer.
  - None of these devices do anything really interesting by themselves, only when working together do interesting things happen.
  - Pushing the analogy a little further, we can address a key question: What *has* become simple? Using a plug is simple. Just find a socket that matches – in your house, at your job, on the road, pretty much anywhere – plug your device in, and, magic, it works!
- The same is true for Java interfaces, as we will see – while an interface is simple, it satisfies standards, and not only are both sides are complicated, but the interactions between the sides are complicated as well.
- So: Interfaces may be simple, but the situations where they are useful are complicated.
- Finally we answer the question: Why does the topic of interfaces seem complicated in Java? Because the situations where they are useful are complicated.

Philosophy – Part 2

- What about class hierarchies?
  - Many authors suggest that composition (using another class as an instance variable) is better than inheritance. I suggest that a master programmer is open to both and develops a good reason for selecting one over the other in a particular situation based on the characteristics of the situation, not a preconception. While many programmers new to Java or to OOD/OOP tend to use inheritance inappropriately, in my opinion, the rule really should be: choose structures that are appropriate to the problem at hand.
  - The simplest rule is something like this: if the application has a number of classes that are naturally related to each other (whatever that means in the context of the particular application), and those classes have some (normally instance) variables in common, then consider using an umbrella class – a parent of all the related classes, and move the common variables and methods to the parent.
  - Notice that here I have focused on the DATA (instance variables = fields) of the classes.
  - A useful and reasonable application of inheritance can be as few as 3 classes, one parent and two children.
- Abstract classes – when to use them.
  - If one or more methods need to implemented by the children, but there is no natural way for the parent to define them, then using an abstract class is a good idea.
  - The signatures of the methods are defined, the compiler is happy if variables in classes that avail themselves of this class group can use variables with the type of the abstract class (leaving the determination of the actual method to call to run-time), the children implement the appropriate version of the methods, and the parent has not had to define worthless and misleading methods.

Philosophy – Part 3

- The relation between abstract and interface classes.
  - An interface (interface class) defines only a set of public instance methods, and perhaps some constants.
  - An abstract class may define anything a regular class defined, plus some additional abstract methods.
  - An abstract class normally revolves around shared data.
  - An interface class is used for shared methods – in other words, shared algorithms.
- Instantiation
  - Neither an interface nor an abstract class may be instantiated.
  - Both may be used as types for variables.
  - The objects referenced by interface variables must be instantiated from concrete classes – classes which implement the appropriate interfaces and extend abstract classes.
  - In all cases, the concrete class must define the required abstract methods.
- Now, let's begin with an overview of the Java language large-scale structures, and then definitions and examples of abstract classes, then interfaces, with lots of examples.

Structures 1

- Java Language Structures – The language provides a variety of structures to organize code
- Souce code files and jar files
  - A source code file (dot-java) is perhaps the most basic unit. Groups of files can be gathered together into packages and packages can then be grouped together into jar files. For simple projects, the default package is all of the class files within a single directory, which is why in small to medium sized projects putting all of the files into a single directory is a good idea.
  - A jar file is convenient way to put a bunch of compiled (class) files into a single file, which can then be efficiently handled. For example, using a jar file will speed up the processing of an applet when sent over a network. This is because if the class files are not in a single jar file, when an applet starts, each class file will have to be requested individually, which creates rather a lot of network overhead.

Structures 2

- Java Language Structures – The classes within files
  - Each Java source code file has at least one class in it: the public class with exactly the same name as the name of the file
- The file may have:
  - Other private classes outside the public class – these classes will not have direct access to the variables or methods of the public class
  - Inner classes – named classes within the body of the public class (or within one of the private classes). These classes DO have access to instance variables and methods. The implications are complicated.
  - Anonymous inner classes – classes declared and defined in the parameter list of a method. This is convenient, but the full implications are even more complex than regular named inner classes.
  - Import statements, comments and javadoc comments.
- All variables and methods must be inside some class. Variables may also be inside a method inside a class. Neither is allowed to exist in a file outside of a class – this is one way in which Java is quite different than C or C++.

Structures 3

- Java Language Structures – A class may have the following:
- Variables
  - Instance variables, also called dynamic or non-static – often these variables should be private
  - Class variables, also called static – these variables should also be private. The most common variable is a count of the number times this class has been instantiated
  - Named constants - “static final”, all caps by convention
- Methods
  - Instance methods – must be called in the context of a real instance of this class or one of its children at run-time.
  - Static methods – normally called in the context of the class name rather than an instance.
  - Static and instance instantiators – blocks of code that look like methods without names, return types, or parameter lists.

Structures 4

- Java Language Structures – Typical methods:
  - public static void main (String args []) - the very special method in the public class of a file which can be called directly by the JVM.
  - Constructors – the default constructor is only available if no constructors are explicitly defined, otherwise a no-parameter constructor will have to be explicitly defined if one is desired.
  - Setters or mutators – methods will allow the controlled change of variables. Often these methods perform various rationality checks.
  - Getters or accessors – methods which report on the state of an instance or a class. The simple approach is to report on the value of an instance variable, but they could report the state of the instance in a more complex manner.
  - ToString – every class should override this method from Object, it is most convenient for debugging.
  - Factory methods – return a new instance of some class, typically with special initialization processing.
  - Facilitators – pretty much any other method in the class.

Structures 5

- Much of this lecture will be about exploring methods related to data structures and interfaces.
- If a class extends another, various methods will often override the ones from the parent.
  - For example, any class that has java.awt.Component as an ancestor can effectively override paint (Graphics) – which is one way interesting GUI's work. The other common feature of GUI's is implementing various interfaces, which we will address in this set of lectures.
- A comment about main: many classes in a package may have this method without causing any problems, and that one use of main is as a test-driver for its class, and may be left in the package even at delivery time.

Abstract – Definition

- The keyword abstract
  - Class
  - Method
- Bodyless methods
- Extends
  - Implement the abstract methods

```
import java.awt.Point;
import java.awt.Graphics2D;

public abstract class MyShape {
   private Point loc = new Point (10, 20);

   public abstract void drawme (Graphics2D g);

   public int getX () {return loc.x;
   } // end method getX

   public int getY () {return loc.y;
   } // end method getY

} // end class Shape

class MyBox extends MyShape {
   private int width = 30, height = 40;

   public void drawme (Graphics2D g) {
      g.drawRect (getX(), getY(), width, height);
   } // end method drawme

} // end class Shape
```

The keyword abstract can be applied to a class or to a method.

Abstract Class

- **When applied to a class, abstract means that the class cannot be instantiated (new'ed).**
- **An abstract class may or may not have abstract methods.**

Abstract Methods

- **An abstract method must be inside an abstract class.**
- **An abstract method has no body.**

Concrete Class

- **A concrete class is one which may be new'ed, and in this context is a class that extends the abstract class, and implements all of the abstract methods.**

Abstract – Example

- The keyword abstract
  - Class
  - Method
- Bodyless methods
- Extends
  - Implement the abstract methods

```
import java.awt.Point;
import java.awt.Graphics2D;

public abstract class MyShape {

   private Point loc = new Point (10, 20);

   public abstract void drawme (Graphics2D g);

   public int getX () {return loc.x;
   } // end method getX

   public int getY () {return loc.y;
   } // end method getY

} // end class Shape

class MyBox extends MyShape {
   private int width = 30, height = 40;

   public void drawme (Graphics2D g) {
      g.drawRect (getX(), getY(), width, height);
   } // end method drawme

} // end class Shape
```

The keyword abstract can be applied to a class or to a method.

Abstract Class

- **When applied to a class, abstract means that the class cannot be instantiated (new'ed).**
- **An abstract class may or may not have abstract methods.**

Abstract Methods

- **An abstract method must be inside an abstract class.**
- **An abstract method has no body.**

Concrete Class

- **A concrete class is one which may be new'ed, and in this context is a class that extends the abstract class, and implements all of the abstract methods.**

Interfaces – Achitecture

- At least 3 classes involved:
- Declaring class – “public interface X”
  - No variables, all methods are abstract
- Client class - “implements X”
  - Provides body of methods specified by X
- Server class – uses X as a variable type
  - Typically in a data structure
  - References become context for method calls
- Availer class – uses the entire structure effectively

Interfaces Declaration

- The keyword interface
  - Class
  - Variables are all public static final
  - Abstract Methods
  - ALL methods are abstract
- Bodyless methods

```
import java.awt.Graphics2D;
import java.awt.Point;

public interface MyShapeInterface {

   int NUMBER = 12;

   public abstract void drawme (Graphics2D g2);

   public Point getLocation ();

} // end MyShapeInterface
```

The keyword interface can only be applied to a class.
Interface Class

- **Declaring a class an interface tells the compiler that this class will have only abstract methods**
- **The keyword abstract is optional for the methods.**
- **All of the methods in this class must be instance (NOT static).**
- **This class will not have any instance variables.**
- **This class may have public static final variables (effectively class constants).**
- **Interfaces may not be instantiated (new'ed).**

Interfaces Declaration

- The keyword interface
  - Class
  - Variables are all public static final
  - Abstract Methods
  - ALL methods are abstract
- Bodyless methods

```
import java.awt.Graphics2D;
import java.awt.Point;

public interface MyShapeInterface {

   int NUMBER = 12;

   public abstract void drawme (Graphics2D g2);

   public Point getLocation ();

} // end MyShapeInterface
```

The keyword interface can only be applied to a class.
Interface Class

- **Declaring a class an interface tells the compiler that this class will have only abstract methods**
- **The keyword abstract is optional for the methods.**
- **All of the methods in this class must be instance (NOT static).**
- **This class will not have any instance variables.**
- **This class may have public static final variables (effectively class constants).**
- **Interfaces may not be instantiated (new'ed).**

Interfaces Client - 1

- The keyword “implements”
  - Interface Class name
  - Implement IF Methods
  - ALL methods are concrete
- This class may be instantiated
- The client class may have class-specific variables and methods
- May extend any class

```
import java.awt.Graphics2D;
import java.awt.Point;

public class MyRect implements MyShapeInterface {
   int width = 20;
   int height = 30;
   int x = 10;
   int y = 20;

   public MyRect (int px, int py, int pw, int ph) {
      x = px; y = py; width = pw; height = ph;
   } // end 4-int constructor

   public void drawme (Graphics2D g2) {
      g2.drawRect (x, y, width, height);
   } // end drawme, required by interface

   public Point getLocation () {
      return new Point (x, y);
   } // end getLocation, required by interface

} // end method MyRect
```

In this example, MyRect implements the MyShapeInterface interface..
I will say that MyRect is a client of the interface to emphasize the relationship between these two classes.

- **The client class must implement all of the abstract methods declared in the interface.**
- **The signatures of the implementing methods must match the signatures of the methods in the interface.**
- **Methods whose signatures don't match are allowed, they just overload the method name as is standard in Java.**
- **The client class may have any other methods and variables as desired – instance and class variables and methods.**

The example here is not complete – it should have the normal host of methods – constructors, setters, getters, and toString.
So far, we have very little motivation for using an interface –
the client could just provide these methods. The motivation will be in the server part.

Interfaces Client - 1

- The keyword “implements”
  - Interface Class name
  - Implement IF Methods
  - ALL methods are concrete
- This class may be instantiated
- The client class may have class-specific variables and methods
- May extend any class

```
import java.awt.Graphics2D;
import java.awt.Point;

public class MyRect implements MyShapeInterface {
   int width = 20;
   int height = 30;
   int x = 10;
   int y = 20;

   public MyRect (int px, int py, int pw, int ph) {
      x = px; y = py; width = pw; height = ph;
   } // end 4-int constructor

   public void drawme (Graphics2D g2) {
      g2.drawRect (x, y, width, height);
   } // end drawme, required by interface

   public Point getLocation () {
      return new Point (x, y);
   } // end getLocation, required by interface

} // end method MyRect
```

In this example, MyRect implements the MyShapeInterface interface..
I will say that MyRect is a client of the interface to emphasize the relationship between these two classes.

- **The client class must implement all of the abstract methods declared in the interface.**
- **The signatures of the implementing methods must match the signatures of the methods in the interface.**
- **Methods whose signatures don't match are allowed, they just overload the method name as is standard in Java.**
- **The client class may have any other methods and variables as desired – instance and class variables and methods.**

The example here is not complete – it should have the normal host of methods – constructors, setters, getters, and toString.
So far, we have very little motivation for using an interface –
the client could just provide these methods. The motivation will be in the server part.

Interfaces Client - 2

- The keyword “implements”
  - Interface Class name
  - Instance variables
  - Constructor
  - Implement IF Methods
  - ALL methods are concrete
- This class may be instantiated
- The client class may have class-specific variables and methods
- May extend any class

```
import java.awt.Graphics2D;
import java.awt.Point;

public class MyCar implements MyShapeInterface {


   int x, y;
   int xb[] = { 0, 15, 15, 20, 20, 50, 50, 55,
                55, 70, 80, 55, 53, 40, 35, 20,  0};
   int yb[] = {20, 20, 25, 25, 20, 20, 25, 25,
                20, 20,  8,  6,  0,  0, 13, 16, 20};

   public MyCar (int px, int py) {
      x = px; y = py;
   } // end int int constructor

   public void drawme (Graphics2D g2) {
      g2.translate (x, y);
      g2.drawPolyline (xb, yb, xb.length);
      g2.translate (-x, -y);
   } // end drawme, required by interface

   public Point getLocation () {
      return new Point (x, y);
   } // end getLocation, required by interface

} // end method MyRect
```

Here's another class the implements the MyShapeInterface interface.
This one is just for fun, but you can see that this class has its own local variables, and a completely different drawme method.
Again, we should complete the class with a lot of other methods to fit into the Java vision of a useful class.
We will see a hint at how an interface can be useful in the next class, the driver with the main method.
After that, we will look at some more complicated examples and try to get some appreciation for the Java vision of interfaces.

Interfaces Client - 2

- The keyword “implements”
  - Interface Class name
  - Instance variables
  - Constructor
  - Implement IF Methods
  - ALL methods are concrete
- This class may be instantiated
- The client class may have class-specific variables and methods
- May extend any class

```
import java.awt.Graphics2D;
import java.awt.Point;

public class MyCar implements MyShapeInterface {

   int x, y;
   int xb[] = { 0, 15, 15, 20, 20, 50, 50, 55,
                55, 70, 80, 55, 53, 40, 35, 20,  0};
   int yb[] = {20, 20, 25, 25, 20, 20, 25, 25,
                20, 20,  8,  6,  0,  0, 13, 16, 20};

   public MyCar (int px, int py) {
      x = px; y = py;
   } // end int int constructor

   public void drawme (Graphics2D g2) {
      g2.translate (x, y);
      g2.drawPolyline (xb, yb, xb.length);
      g2.translate (-x, -y);
   } // end drawme, required by interface

   public Point getLocation () {
      return new Point (x, y);
   } // end getLocation, required by interface

} // end method MyRect
```

Here's another class the implements the MyShapeInterface interface.
This one is just for fun, but you can see that this class has its own local variables, and a completely different drawme method.
Again, we should complete the class with a lot of other methods to fit into the Java vision of a useful class.
We will see a hint at how an interface can be useful in the next class, the driver with the main method.
After that, we will look at some more complicated examples and try to get some appreciation for the Java vision of interfaces.

Interfaces Availer*

- Program uses an interface
  - Interface array variable
  - Instantiate concrete classes
  - Loop over array
  - Use IF Methods
- Graphics and main stuff

```
import javax.swing.JFrame;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Color;

public class MyMain extends JFrame {
   static final long serialVersionUID =
                        -4560888949952421 30L;

   MyShapeInterface [] ma = {new MyRect (50, 225, 300, 7),
                             new MyCar ( 75, 200),
                             new MyCar (250, 200)};

   public void paint (Graphics g) {
      super.paint(g);
      Graphics2D g2 = (Graphics2D) g;

      for (MyShapeInterface mc : ma)

        mc.drawme (g2);

   } // end standard paint

   public static void main (String args []) {
      MyMain mm = new MyMain ();
      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.green);
      mm.setTitle ("The Application");
      mm.setSize (400,400);
      mm.setVisible(true);
   } // end main

} // end class MyMain
```

Here is a class that tests the classes created so far.
This is a fairly interesting use of an interface.

Interface Features:

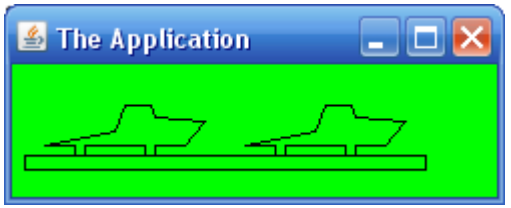
- **> The type of the array ma is an interface.**
- **> The elements of the array only implement the interface.**
- **> The classes of the elements of the array don't extend any particular class.**
- **> The type of mc is also an interface**
- **> The compiler accepts drawme through the interface**

Graphics issues:
The class creates a window and draws cars and a box.
The paint method is used to connect to Jframe.
The super call and the cast are details for another presentation.
The serialVersion is because JFrame implements Serializable.

*** - Availer is not standard notation, but the user of an interface system (interface/client/server) Deserves its own name.**

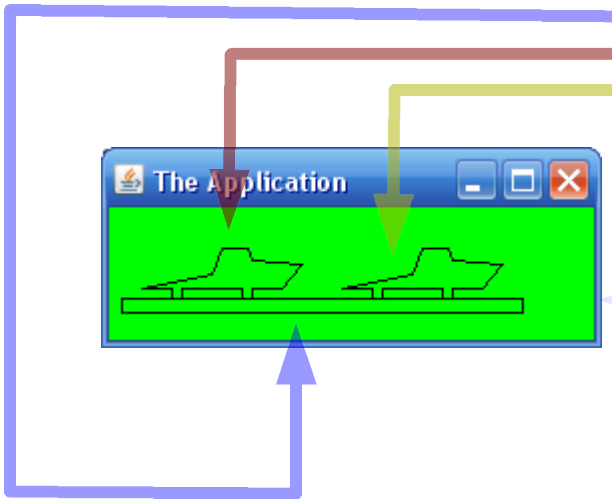
Interfaces Availer

- Program uses an interface
  - Interface array variable
  - Instantiate concrete classes
  - Loop over array
  - Use IF Methods
- Graphics and main stuff




```
import javax.swing.JFrame;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Color;

public class MyMain extends JFrame {
   static final long serialVersionUID =
                        -4560888949952421 30L;

   MyShapeInterface [] ma = {new MyRect (10, 75, 200, 7),
                             new MyCar ( 20, 50),
                             new MyCar (120, 50)};

   public void paint (Graphics g) {
      super.paint(g);
      Graphics2D g2 = (Graphics2D) g;

      for (MyShapeInterface mc : ma)

        mc.drawme (g2);

   } // end standard paint

   public static void main (String args []) {
      MyMain mm = new MyMain ();
      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.green);
      mm.setTitle ("The Application");
      mm.setSize (250, 100);
      mm.setVisible(true);
   } // end main

} // end class MyMain
```

Interfaces Availer




```
import javax.swing.JFrame;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Color;

public class MyMain extends JFrame {
   static final long serialVersionUID =
                        -4560888949952421 30L;

   MyShapeInterface [] ma = {new MyRect (10, 75, 200, 7),
                             new MyCar ( 20, 50),
                             new MyCar (120, 50)};

   public void paint (Graphics g) {
      super.paint(g);
      Graphics2D g2 = (Graphics2D) g;

      for (MyShapeInterface mc : ma)

        mc.drawme (g2);

   } // end standard paint

   public static void main (String args []) {
      MyMain mm = new MyMain ();
      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.green);
      mm.setTitle ("The Application");
      mm.setSize (250, 100);
      mm.setVisible(true);
   } // end main

} // end class MyMain
```

Common Interfaces with Examples

- Events – EventListener, ActionListener
- Collections – Collection, List, Queue, Iterable *
- Order – Comparable, Comparator *
- I/O – Serializable **
- Threads – Runnable **

* - Part 2

** - Part 3

Events Example – 1a

This example shows a GUI implementing an interface, and using the connecting the interface method.

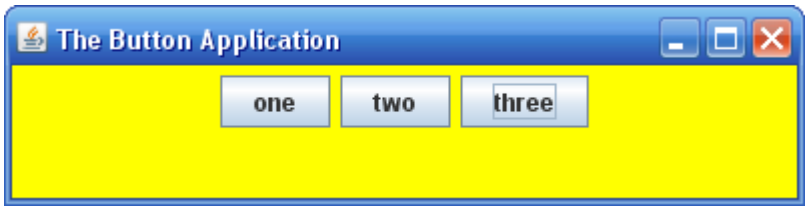
This class will put up 3 buttons, and when a button is pressed, it will bring up a message with details of the event. See next slide for the display.

The next few slides will point out a few of the more interesting features of this program.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyMainB extends JFrame
       implements ActionListener {

   JButton [] bt =
      {new JButton ("one"),
       new JButton ("two"),
       new JButton ("three")};
```

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

      MyMainB mm = new MyMainB ();

      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.yellow);
      mm.setLayout (new FlowLayout ());
      mm.setTitle ("The Button Application");
      mm.setSize (400,100);

      mm.add (mm.bt[0]);
      mm.add (mm.bt[1]);
      mm.add (mm.bt[2]);

      mm.bt[0].addActionListener (mm);
      mm.bt[1].addActionListener (mm);
      mm.bt[2].addActionListener (mm);

      mm.setVisible(true);
   } // end main

   public void actionPerformed (ActionEvent e) {
      JOptionPane.showMessageDialog ( null,
         "You pressed:\n\n" +
             e.toString().replace(',', '\n'),
         "The Answer", JOptionPane.INFORMATION_MESSAGE);
   } // end method actionPerformed

    static final long serialVersionUID =
                          -7353858170081248L;
} // end class MyMain
```

Events Example – 1b



This section imports the relevant classes:
> 3 from swing
> 2 display (FlowLayout and Color)
> 2 related to the event handling interface

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyMainB extends JFrame
       implements ActionListener {

   JButton [] bt =
      {new JButton ("one"),
       new JButton ("two"),
       new JButton ("three")};

   public static void main (String args []) {

      MyMainB mm = new MyMainB ();

      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.yellow);
      mm.setLayout (new FlowLayout ());
      mm.setTitle ("The Button Application");
      mm.setSize (400,100);

      mm.add (mm.bt[0]);
      mm.add (mm.bt[1]);
      mm.add (mm.bt[2]);

      mm.bt[0].addActionListener (mm);
      mm.bt[1].addActionListener (mm);
      mm.bt[2].addActionListener (mm);

      mm.setVisible(true);
   } // end main

   public void actionPerformed (ActionEvent e) {
      JOptionPane.showMessageDialog ( null,
         "You pressed:\n\n" +
             e.toString().replace(',', '\n'),
         "The Answer", JOptionPane.INFORMATION_MESSAGE);
   } // end method actionPerformed

    static final long serialVersionUID =
                          -7353858170081248L;
} // end class MyMain
```

Events Example – 1c

The declaration of the class extends Jframe so it can be drawn by the JVM, and implements the interface ActionListener.

The imlementation is what allows this class to react to button presses, as will be explained shortly.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyMainB extends JFrame
       implements ActionListener {

   JButton [] bt =
      {new JButton ("one"),
       new JButton ("two"),
       new JButton ("three")};

   public static void main (String args []) {

      MyMainB mm = new MyMainB ();

      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.yellow);
      mm.setLayout (new FlowLayout ());
      mm.setTitle ("The Button Application");
      mm.setSize (400,100);

      mm.add (mm.bt[0]);
      mm.add (mm.bt[1]);
      mm.add (mm.bt[2]);

      mm.bt[0].addActionListener (mm);
      mm.bt[1].addActionListener (mm);
      mm.bt[2].addActionListener (mm);

      mm.setVisible(true);
   } // end main

   public void actionPerformed (ActionEvent e) {
      JOptionPane.showMessageDialog ( null,
         "You pressed:\n\n" +
             e.toString().replace(',', '\n'),
         "The Answer", JOptionPane.INFORMATION_MESSAGE);
   } // end method actionPerformed

    static final long serialVersionUID =
                        -7353858170081248L;
} // end class MyMain
```

Events Example – 1d

This next part declares an array of buttons, and instatiates each of them.

It is worth noting what this section DOES NOT do:
> it does not put the buttons into any display
> it does not make the buttons active

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyMainB extends JFrame
       implements ActionListener {

   JButton [] bt =
      {new JButton ("one"),
       new JButton ("two"),
       new JButton ("three")};

   public static void main (String args []) {

      MyMainB mm = new MyMainB ();

      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.yellow);
      mm.setLayout (new FlowLayout ());
      mm.setTitle ("The Button Application");
      mm.setSize (400,100);

      mm.add (mm.bt[0]);
      mm.add (mm.bt[1]);
      mm.add (mm.bt[2]);

      mm.bt[0].addActionListener (mm);
      mm.bt[1].addActionListener (mm);
      mm.bt[2].addActionListener (mm);

      mm.setVisible(true);
   } // end main

   public void actionPerformed (ActionEvent e) {
      JOptionPane.showMessageDialog ( null,
         "You pressed:\n\n" +
             e.toString().replace(',', '\n'),
         "The Answer", JOptionPane.INFORMATION_MESSAGE);
   } // end method actionPerformed

    static final long serialVersionUID =
                        -7353858170081248L;
} // end class MyMain
```

Events Example – 1e

The methods:

> main – public static void

> actionPerformed – public void
THIS is the method specified by the interface

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyMainB extends JFrame
       implements ActionListener {

   JButton [] bt =
      {new JButton ("one"),
       new JButton ("two"),
       new JButton ("three")};
```

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

      MyMainB mm = new MyMainB ();

      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.yellow);
      mm.setLayout (new FlowLayout ());
      mm.setTitle ("The Button Application");
      mm.setSize (400,100);

      mm.add (mm.bt[0]);
      mm.add (mm.bt[1]);
      mm.add (mm.bt[2]);

      mm.bt[0].addActionListener (mm);
      mm.bt[1].addActionListener (mm);
      mm.bt[2].addActionListener (mm);

      mm.setVisible(true);
   } // end main
```

```
   public void actionPerformed (ActionEvent e) {
      JOptionPane.showMessageDialog ( null,
         "You pressed:\n\n" +
             e.toString().replace(',', '\n'),
         "The Answer", JOptionPane.INFORMATION_MESSAGE);
   } // end method actionPerformed
```

```
    static final long serialVersionUID =
                          -7353858170081248L;
} // end class MyMain
```

Events Example – 1f

In the main method

Setting up the GUI environment

Adding the buttons to the display

Connecting the buttons to the event handling mechanism of the Java Virtual Machine. That mechanism has a data structure of objects that implement event handlers, as MyMainB does, and when an event happens, it calls the actionPerformed method. Here, all the buttons do the same thing.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyMainB extends JFrame
       implements ActionListener {

   JButton [] bt =
      {new JButton ("one"),
       new JButton ("two"),
       new JButton ("three")};

   public static void main (String args []) {

      MyMainB mm = new MyMainB ();

      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.yellow);
      mm.setLayout (new FlowLayout ());
      mm.setTitle ("The Button Application");
      mm.setSize (400,100);

      mm.add (mm.bt[0]);
      mm.add (mm.bt[1]);
      mm.add (mm.bt[2]);

      mm.bt[0].addActionListener (mm);
      mm.bt[1].addActionListener (mm);
      mm.bt[2].addActionListener (mm);

      mm.setVisible(true);
   } // end main

   public void actionPerformed (ActionEvent e) {
      JOptionPane.showMessageDialog ( null,
         "You pressed:\n\n" +
             e.toString().replace(',', '\n'),
         "The Answer", JOptionPane.INFORMATION_MESSAGE);
   } // end method actionPerformed

    static final long serialVersionUID =
                          -7353858170081248L;
} // end class MyMain
```

Events Example – 1g

In the actionPerformed method

When one of the three buttons is pressed, an event is generated and the JVM determines, through the registered listeners, that this class has a method that should be called. That method, actionPerformed, is known because it is an interface method.

In this case, the code just displays a message box with a toString representation of the event. This is a pretty involved display. The next examples will explore customizing the response to the buttons.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyMainB extends JFrame
       implements ActionListener {

   JButton [] bt =
      {new JButton ("one"),
       new JButton ("two"),
       new JButton ("three")};
```

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

      MyMainB mm = new MyMainB ();

      mm.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
      mm.getContentPane().setBackground (Color.yellow);
      mm.setLayout (new FlowLayout ());
      mm.setTitle ("The Button Application");
      mm.setSize (400,100);

      mm.add (mm.bt[0]);
      mm.add (mm.bt[1]);
      mm.add (mm.bt[2]);

      mm.bt[0].addActionListener (mm);
      mm.bt[1].addActionListener (mm);
      mm.bt[2].addActionListener (mm);

      mm.setVisible(true);
   } // end main

   public void actionPerformed (ActionEvent e) {
      JOptionPane.showMessageDialog ( null,
         "You pressed:\n\n" +
             e.toString().replace(',', '\n'),
         "The Answer", JOptionPane.INFORMATION_MESSAGE);
   } // end method actionPerformed

    static final long serialVersionUID =
                        -7353858170081248L;
} // end class MyMain
```

Events Example – 2a

```
public class MyMainC extends JFrame {

   JButton [] bt = {new JButton ("0"),
                    new JButton ("0"),
                    new JButton ("Clear")};

   int [] bv = new int [bt.length - 1];

   public MyMainC () {
      setDefaultCloseOperation
             (JFrame.EXIT_ON_CLOSE);
      getContentPane().setBackground
             (Color.yellow);
      setLayout (new FlowLayout ());
      setTitle ("The Button Application");
      setSize (400,100);

      add (bt[0]); add (bt[1]); add (bt[2]);

      bt[0].addActionListener
            (new MyButtonZero ());
      bt[1].addActionListener
            (new MyButtonOne  ());
      bt[2].addActionListener
            (new MyButtonTwo  ());

      setVisible(true);
   } // end no-parameter constructor
```

The imports are the same as before, and we are running out of space for code and comments. This is one approach to having the buttons do interesting things.

```
   public static void main (String args []) {
      MyMainC mm = new MyMainC ();
   } // end main

   class MyButtonZero implements ActionListener {
      public void actionPerformed (ActionEvent e) {
         bv [0] ++; bt[0].setText ("" + bv[0]);
      } // end method actionPerformed
   } // end class MyButtonOne

   class MyButtonOne implements ActionListener {
      public void actionPerformed (ActionEvent e) {
         bv [1] ++; bt[1].setText ("" + bv[1]);
      } // end method actionPerformed
   } // end class MyButtonOne

   class MyButtonTwo implements ActionListener {
      public void actionPerformed (ActionEvent e) {
         bv [0] = 0; bt[0].setText ("" + bv[0]);
         bv [1] = 0; bt[1].setText ("" + bv[1]);
      } // end method actionPerformed
   } // end class MyButtonOne

    static final long serialVersionUID =
-7353858170081248L;
} // end class MyMainC
```

The code was moved to the constructor to let let the addActionListeners create and reference instances of the inner classes created to handle events from particular buttons.

Events Example – 2b

```
public class MyMainC extends JFrame {

   JButton [] bt = {new JButton ("0"),
                    new JButton ("0"),
                    new JButton ("Clear")};

   int [] bv = new int [bt.length - 1];

   public MyMainC () {
      setDefaultCloseOperation
              (JFrame.EXIT_ON_CLOSE);
      getContentPane().setBackground
              (Color.yellow);
      setLayout (new FlowLayout ());
      setTitle ("The Button Application");
      setSize (400,100);

      add (bt[0]); add (bt[1]); add (bt[2]);

      bt[0].addActionListener
            (new MyButtonZero ());
      bt[1].addActionListener
            (new MyButtonOne  ());
      bt[2].addActionListener
            (new MyButtonTwo  ());

      setVisible(true);
   } // end no-parameter constructor
```

The default constructor
Configuring the Listeners
Adding the buttons

```
   public static void main (String args []) {
      MyMainC mm = new MyMainC ();
   } // end main

   class MyButtonZero implements ActionListener {
      public void actionPerformed (ActionEvent e) {
         bv [0] ++; bt[0].setText ("" + bv[0]);
      } // end method actionPerformed
   } // end class MyButtonOne

   class MyButtonOne implements ActionListener {
      public void actionPerformed (ActionEvent e) {
         bv [1] ++; bt[1].setText ("" + bv[1]);
      } // end method actionPerformed
   } // end class MyButtonOne

   class MyButtonTwo implements ActionListener {
      public void actionPerformed (ActionEvent e) {
         bv [0] = 0; bt[0].setText ("" + bv[0]);
         bv [1] = 0; bt[1].setText ("" + bv[1]);
      } // end method actionPerformed
   } // end class MyButtonOne

    static final long serialVersionUID =
-7353858170081248L;
} // end class MyMainC
```

Each Listener has its own class in this approach.

Events Example – 2c

```
public class MyMainC extends JFrame {

   JButton [] bt = {new JButton ("0"),
                    new JButton ("0"),
                    new JButton ("Clear")};

   int [] bv = new int [bt.length - 1];

   public MyMainC () {
      setDefaultCloseOperation
             (JFrame.EXIT_ON_CLOSE);
      getContentPane().setBackground
             (Color.yellow);
      setLayout (new FlowLayout ());
      setTitle ("The Button Application");
      setSize (400,100);

      add (bt[0]); add (bt[1]); add (bt[2]);

      bt[0].addActionListener
           (new MyButtonZero ());

      bt[1].addActionListener
           (new MyButtonOne  ());

      bt[2].addActionListener
           (new MyButtonTwo  ());

      setVisible(true);
   } // end no-parameter constructor

   public static void main (String args []) {
      MyMainC mm = new MyMainC ();
   } // end main

   class MyButtonZero implements ActionListener {
      public void actionPerformed (ActionEvent e) {
         bv [0] ++; bt[0].setText ("" + bv[0]);
      } // end method actionPerformed
   } // end class MyButtonOne

   class MyButtonOne implements ActionListener {
      public void actionPerformed (ActionEvent e) {
         bv [1] ++; bt[1].setText ("" + bv[1]);
      } // end method actionPerformed
   } // end class MyButtonOne

   class MyButtonTwo implements ActionListener {
      public void actionPerformed (ActionEvent e) {
         bv [0] = 0; bt[0].setText ("" + bv[0]);
         bv [1] = 0; bt[1].setText ("" + bv[1]);
      } // end method actionPerformed
   } // end class MyButtonOne

    static final long serialVersionUID =
-7353858170081248L;
} // end class MyMainC
```

These are the correspondences between the addActionListener and the classes.

Notice how each button is tied to the corresponding class through the addActionListener method.

The counting array, bv, is an instance array of MyMainC, the inner classes see those variables, and they are used to change the labels on the buttons.

Events Example – 3a

```
public class MyMainD extends JFrame {

   MyJButton [] bt = new MyJButton [5];

   public MyMainD (String title, Color bgc) {
      setDefaultCloseOperation
           (JFrame.EXIT_ON_CLOSE);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setTitle (title);
      setSize (400,100);

      for (MyJButton mb : bt) {
         mb = new MyJButton (); // mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor

   public static void main (String args []) {
      MyMainD ma =
          new MyMainD ("Blue", Color.blue);
      MyMainD mb =
          new MyMainD ("Green", Color.green);
   } // end main

    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMain
```

```
class MyJButton extends JButton
      implements ActionListener {

   int count = 0;

   public MyJButton () {
      setText ("0");
      addActionListener(this);
   } // end no parameter constructor

   public void actionPerformed
               (ActionEvent e) {
      setText ("" + ++count);
   } // end method actionPerformed

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyButtonOne
```

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

These are the imports require to make this program work. They will not be repeated.

Here is another way to implement the same kind of example -
A few buttons in a frame.
In this case, the main method instantiates 2 windows, with 5 buttons each.
The 5 buttons come from the instance variable bt – a 5 element array.
Notice how simple the main program is!

The constructor accepts the window title and background color as parameters.

Events Example – 3b

```
public class MyMainD extends JFrame {

   MyJButton [] bt = new MyJButton [5];


   public MyMainD (String title, Color bgc) {
      setDefaultCloseOperation
           (JFrame.EXIT_ON_CLOSE);
      setTitle (title);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setSize (400,100);

      for (MyJButton mb : bt) {
         mb = new MyJButton (); // mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor


   public static void main (String args []) {
      MyMainD ma =
          new MyMainD ("Blue", Color.blue);
      MyMainD mb =
          new MyMainD ("Green", Color.green);
   } // end main


    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMain
```

```
class MyJButton extends JButton
      implements ActionListener {

   int count = 0;

   public MyJButton () {
      setText ("0");
      addActionListener(this);
   } // end no parameter constructor

   public void actionPerformed
               (ActionEvent e) {
      setText ("" + ++count);
   } // end method actionPerformed

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyButtonOne
```

The two windows.
Note that all of the buttons are independent of each other.

Events Example – 3c

```
public class MyMainD extends JFrame {

   MyJButton [] bt = new MyJButton [5];

   public MyMainD (String title, Color bgc) {
      setDefaultCloseOperation
           (JFrame.EXIT_ON_CLOSE);
      setTitle (title);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setSize (400,100);

      for (MyJButton mb : bt) {
         mb = new MyJButton (); // mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor

   public static void main (String args []) {
      MyMainD ma =
          new MyMainD ("Blue", Color.blue);
      MyMainD mb =
          new MyMainD ("Green", Color.green);
   } // end main

    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMain
```

```
class MyJButton extends JButton
      implements ActionListener {

   int count = 0;

   public MyJButton () {
      setText ("0");
      addActionListener(this);
   } // end no parameter constructor

   public void actionPerformed
                (ActionEvent e) {
      setText ("" + ++count);
   } // end method actionPerformed

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyButtonOne
```

Here is another way to implement the same kind of example -

A few buttons in a frame.

The 5 buttons come from the instance variable bt – a 5 element array.

The constructor accepts the window title and background color as parameters.

In this case, the main method instantiates 2 windows, with 5 buttons each.

Notice how simple the main program is – just 2 lines of code!

Events Example – 3d

```
public class MyMainD extends JFrame {

   MyJButton [] bt = new MyJButton [5];


   public MyMainD (String title, Color bgc) {
      setDefaultCloseOperation
           (JFrame.EXIT_ON_CLOSE);
      setTitle (title);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setSize (400,100);

      for (MyJButton mb : bt) {
         mb = new MyJButton (); // mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor


   public static void main (String args []) {
      MyMainD ma =
          new MyMainD ("Blue", Color.blue);
      MyMainD mb =
          new MyMainD ("Green", Color.green);
   } // end main


    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMain
```

```
class MyJButton extends JButton
      implements ActionListener {

   int count = 0;

   public MyJButton () {
      setText ("0");
      addActionListener(this);
   } // end no parameter constructor

   public void actionPerformed
               (ActionEvent e) {
      setText ("" + ++count);
   } // end method actionPerformed

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyButtonOne
```

This iterator for loop is quite clever, and has little extraneous syntax.

The size of the loop is the size of the array bt.

The loop variable mb is now shorthand for an array element – with no explicit index!

Before the “new”, each entry in the array was null.

Instantiation is interesting, but more on that next slide.

Finally each button is added to this instance of MyMainD – a JFrame..

In this program, main instantiates two frames of 5 buttons each.

Events Example – 3e

```
public class MyMainD extends JFrame {

   MyJButton [] bt = new MyJButton [5];


   public MyMainD (String title, Color bgc) {
      setDefaultCloseOperation
           (JFrame.EXIT_ON_CLOSE);
      setTitle (title);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setSize (400,100);

      for (MyJButton mb : bt) {
         mb = new MyJButton (); // mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor


   public static void main (String args []) {
      MyMainD ma =
          new MyMainD ("Blue", Color.blue);
      MyMainD mb =
          new MyMainD ("Green", Color.green);
   } // end main


    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMain
```

```
class MyJButton extends JButton
      implements ActionListener {

   int count = 0;

   public MyJButton () {
      setText ("0");
      addActionListener(this);
   } // end no parameter constructor

   public void actionPerformed
                (ActionEvent e) {
      setText ("" + ++count);
   } // end method actionPerformed

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyButtonOne
```

This class, MyJButton, may be in its own file or a private class within the MyMainD file.

Method required by implements declaration –
actionPerformed (ActionEvent)
This method changes its own instance of count and sets its own text – so the button label increments with every click.

Constructor – no parameter
setText uses AbstractButton version
addActionListener uses AbstractButton version

Instance variable: count
Each instance of MyJButton has its own copy of this variable, automatically.

Events Example – 4a

```
public class MyMainE extends JFrame {

   MyJButton2 [] bt = new MyJButton2 [5];

   public MyMainE (String title, Color bgc) {
      setDefaultCloseOperation
         (JFrame.EXIT_ON_CLOSE);
      setTitle (title);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setSize (400,100);

      for (MyJButton2 mb : bt) {
         mb = new MyJButton2 (); //mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor

   public static void main (String args []) {
      MyMainE ma =
         new MyMainE ("Blue", Color.blue);
   } // end main

    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMain
```

```
class MyJButton2 extends JButton {

   public MyJButton2 () {
      setText ("Help!");
      addActionListener(new MyCry ());
   } // end no parameter constructor

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyButtonOne

class MyCry implements ActionListener {

   static String st = "Help!\n" +
      "I'm trapped in this button\n" +
      "And can't escape!\n";

   int count = 0;

   public void actionPerformed
           (ActionEvent e) {
      JoptionPane.showMessageDialog
         (null,
          st + ++count,
          "Help! Help!",
          JOptionPane.WARNING_MESSAGE);
   } // end method actionPerformed

} // end class MyCry
```

In this example, a completely different class, MyCry, implements ActionListener – again with its own local variable. MyJButton2 does not implement that interface. This example emphasizes that the listener is not required to be related to any other classes in the program.

The imports are the same as those in example 3, and MyMainE is almost identical to MyMainD.

Events Example – 4b

```
public class MyMainE extends JFrame {

   MyJButton2 [] bt = new MyJButton2 [5];

   public MyMainE (String title, Color bgc) {
      setDefaultCloseOperation
         (JFrame.EXIT_ON_CLOSE);
      setTitle (title);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setSize (400,100);

      for (MyJButton2 mb : bt) {
         mb = new MyJButton2 (); //mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor

   public static void main (String args []) {
      MyMainE ma =
         new MyMainE ("Blue", Color.blue);
   } // end main

    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMain
```

```
class MyJButton2 extends JButton {

   public MyJButton2 () {
      setText ("Help!");
      addActionListener(new MyCry ());
   } // end no parameter constructor

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyButtonOne

class MyCry implements ActionListener {

   static String st = "Help!\n" +
      "I'm trapped in this button\n" +
      "And can't escape!\n";

   int count = 0;

   public void actionPerformed
         (ActionEvent e) {
      JoptionPane.showMessageDialog
         (null,
          st + ++count,
          "Help! Help!",
          JOptionPane.WARNING_MESSAGE);
   } // end method actionPerformed

} // end class MyCry
```

The code in MyMainE is based on the code in MyMainD. The differences are:

(a) the code uses MyJButton2 rather than MyJButton

(b) only one copy of MyMainE is instantiated in main

Events Example – 4c

```
public class MyMainE extends JFrame {

   MyJButton2 [] bt = new MyJButton2 [5];

   public MyMainE (String title, Color bgc) {
      setDefaultCloseOperation
         (JFrame.EXIT_ON_CLOSE);
      setTitle (title);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setSize (400,100);

      for (MyJButton2 mb : bt) {
         mb = new MyJButton2 (); //mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor

   public static void main (String args []) {
      MyMainE ma =
         new MyMainE ("Blue", Color.blue);
   } // end main

    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMain
```

```
class MyJButton2 extends JButton {

   public MyJButton2 () {
      setText ("Help!");
      addActionListener(new MyCry ());
   } // end no parameter constructor

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyButtonOne

class MyCry implements ActionListener {

   static String st = "Help!\n" +
      "I'm trapped in this button\n" +
      "And can't escape!\n";

   int count = 0;

   public void actionPerformed
           (ActionEvent e) {
      JoptionPane.showMessageDialog
         (null,
          st + ++count,
          "Help! Help!",
          JOptionPane.WARNING_MESSAGE);
   } // end method actionPerformed

} // end class MyCry
```

The class MyJButton2 no longer has an instance variable, no longer implements ActionListener, and no longer defines the method ActionPerformed.

It still does one very interesting thing – it creates a new instance of a completely unrelated class, MyCry, and configures that instance to be given events generated by button presses.

Events Example – 4d

```
public class MyMainE extends JFrame {

   MyJButton2 [] bt = new MyJButton2 [5];

   public MyMainE (String title, Color bgc) {
      setDefaultCloseOperation
         (JFrame.EXIT_ON_CLOSE);
      setTitle (title);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setSize (400,100);

      for (MyJButton2 mb : bt) {
         mb = new MyJButton2 (); //mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor

   public static void main (String args []) {
      MyMainE ma =
         new MyMainE ("Blue", Color.blue);
   } // end main

    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMainE
```

```
class MyJButton2 extends JButton {

   public MyJButton2 () {
      setText ("Help!");
      addActionListener(new MyCry ());
   } // end no parameter constructor

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyJButton2

class MyCry implements ActionListener {
   static String st = "Help!\n" +
      "I'm trapped in this button\n" +
      "And can't escape!\n";
   int count = 0;

   public void actionPerformed
         (ActionEvent e) {
      JoptionPane.showMessageDialog
         (null,
          st + ++count,
          "Help! Help!",
          JOptionPane.WARNING_MESSAGE);
   } // end method actionPerformed

} // end class MyCry
```

Exercise: Trace through this code to make sure you understand what classes are instantiated, where, and how the run-time connections are made.

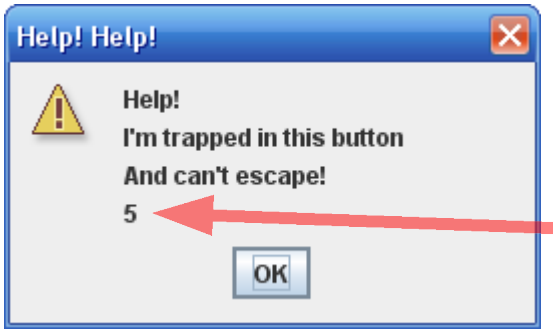
Start with main, then instantiate a copy of MyMainE, go through the constructor, create instances of MyJButton2, create a new instance of MyCry, activate by using addActionListener, then add the new button to the display.

Finally, trace what happens when one of the buttons is pressed. And make sure you understand how the dialogs change when pressing the different buttons.

Events Example – 4d

```
public class MyMainE extends JFrame {

   MyJButton2 [] bt = new MyJButton2 [5];

   public MyMainE (String title, Color bgc) {
      setDefaultCloseOperation
         (JFrame.EXIT_ON_CLOSE);
      setTitle (title);
      getContentPane().setBackground (bgc);
      setLayout (new FlowLayout ());
      setSize (400,100);

      for (MyJButton2 mb : bt) {
         mb = new MyJButton2 (); //mb was null
         add(mb);
      } // end for each entry in bt

      setVisible(true);
   } // end no-parameter constructor

   public static void main (String args []) {
      MyMainE ma =
         new MyMainE ("Blue", Color.blue);
   } // end main

    static final long serialVersionUID =
      -7353858170081248L;
} // end class MyMainE
```

```
class MyJButton2 extends JButton {

   public MyJButton2 () {
      setText ("Help!");
      addActionListener(new MyCry ());
   } // end no parameter constructor

   static final long serialVersionUID =
      5975214558720418398L;
} // end class MyJButton2

class MyCry implements ActionListener {
   static String st = "Help!\n" +
      "I'm trapped in this button\n" +
      "And can't escape!\n";
   int count = 0;

   public void actionPerformed
          (ActionEvent e) {
      JoptionPane.showMessageDialog
         (null,
          st + ++count,
          "Help! Help!",
          JOptionPane.WARNING_MESSAGE);
   } // end method actionPerformed

} // end class MyCry
```



The displays
Note the count in the message dialog box.