Object Orientation with Design Patterns

Chain of Responsibility
Observer
Iterator

Chain of Responsibility

- **Intent:**
  Avoid coupling the sender of a request to its receiver by giving **more than one object a chance to handle the request**. Chain the receiving objects and **pass the request along the chain** until an object handles it.

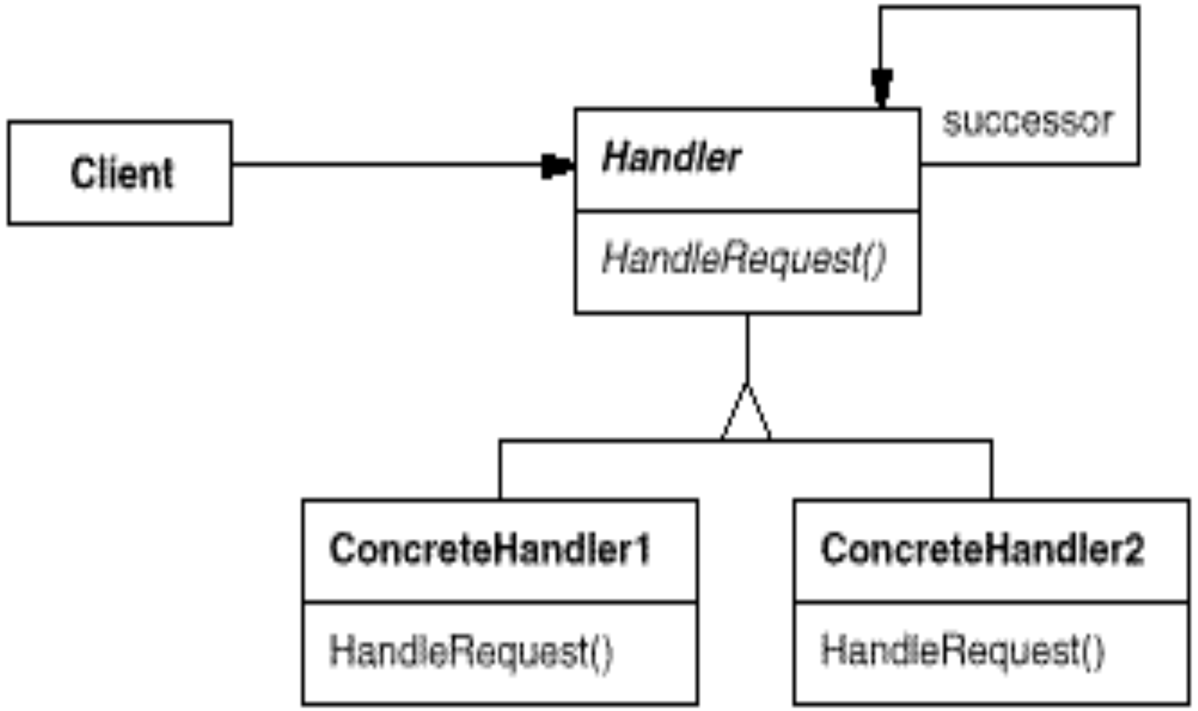
- The Chain of Responsibility pattern **allows a number of classes to attempt to handle a request** without any of them **knowing** about the **capabilities** of the other objects.

- It provides **loose coupling between these classes**; the only **common link** is the request that is passed between them. The request is passed along until **one of the classes can handle it.**

Applicability

- The Chain of responsibility is a good example of a pattern that helps to **keep separate the knowledge** of what each object in a program can do. That is, it **reduces the coupling** between objects so that they can **act independantly**. You will find this pattern helpful when :
  - It is more appropriate for **the objects to decide which one is to carry out the request** than it is for you to build the decision into your code.
  - There might be **new objects** that you want to add to the list of processing options while the program is executing.
  - Sometimes **more than one object** will have to act on a request, and you **don't want to build knowledge** of these interactions into the calling program.

Structure



Client
Handler
HandleRequest()
successor
ConcreteHandler1
HandleRequest()
ConcreteHandler2
HandleRequest()

Participants

- **Handler**
  - **defines an interface for handling requests**.
  - (optional) implements the successor link.
- **ConcreteHandler**
  - **handles requests it is responsible for**.
  - can access its successor.
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
- **Client**
  - **initiates the request to a ConcreteHandler object on the chain**.

Web Examples

- http://www.codeproject.com/Articles/41786/Chain-of-Responsibility-Design-Pattern

Chain of Responsibility

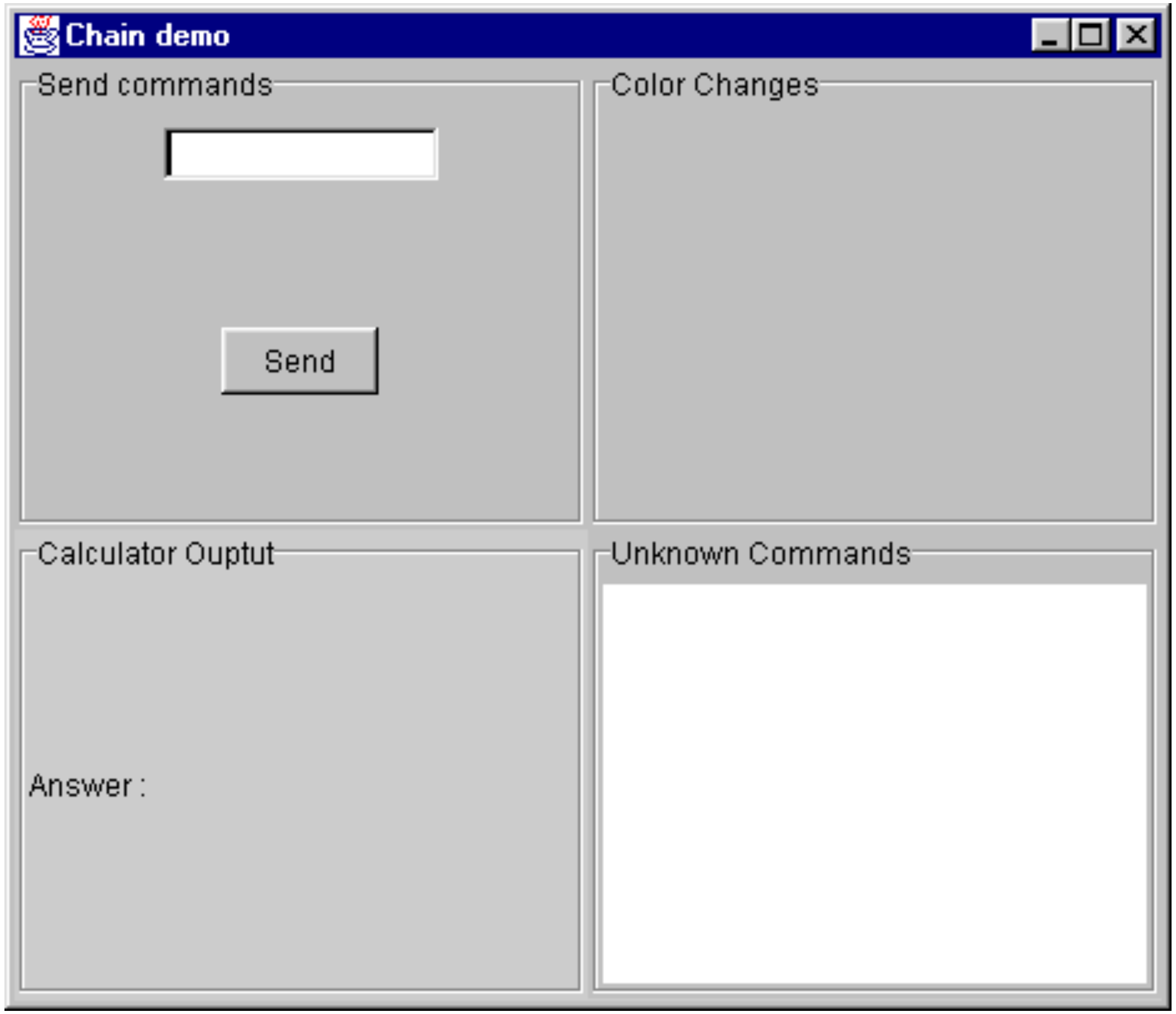
- Consider the following program. It is comprised of **four separate sections**. In the first section the user can type a text command and press the send button.

- If the command is a **color** such as ‘red’, ‘green’, ‘blue’ the color panel will change color.

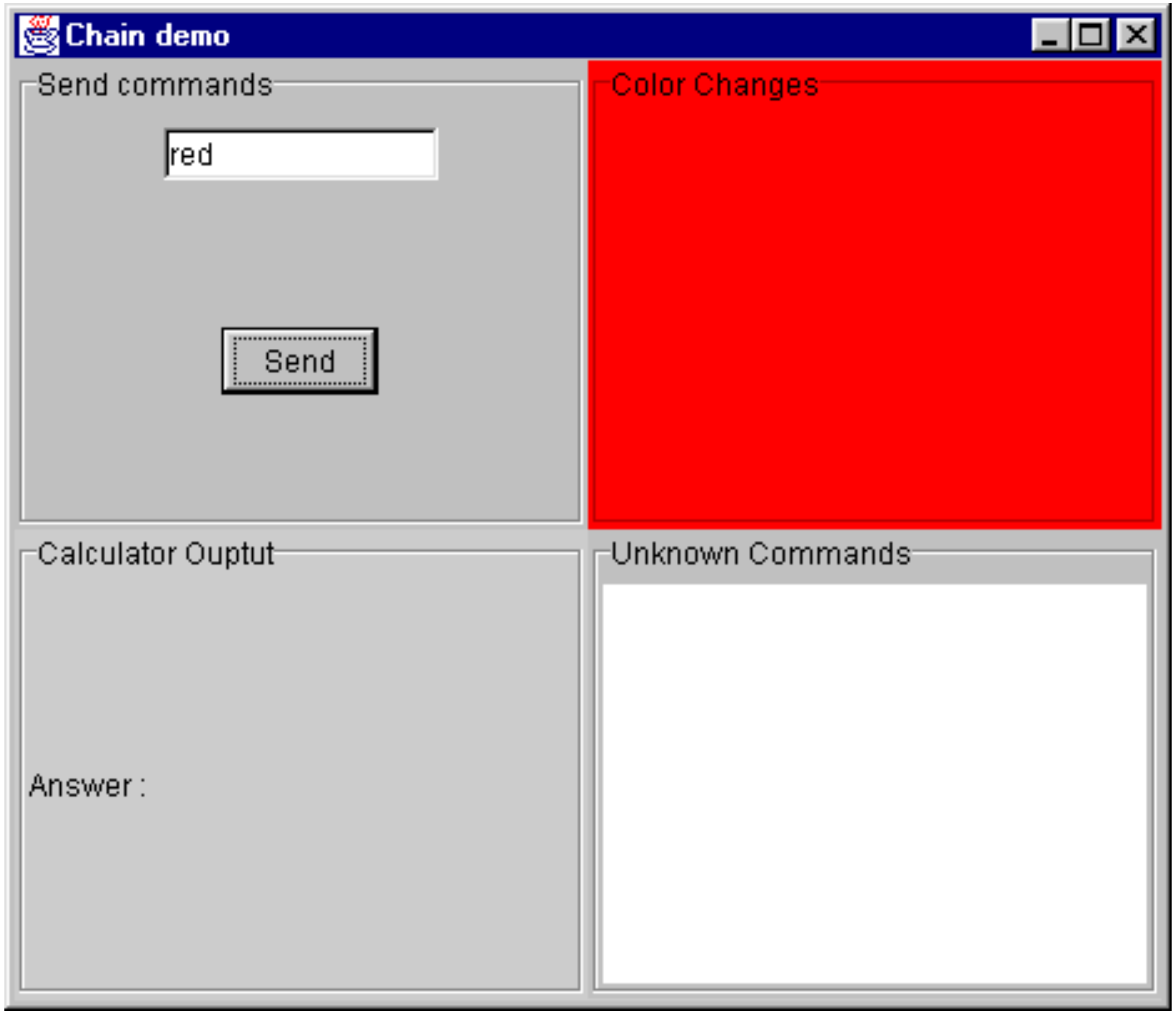
- If the command is a **simple calculation** such as ‘1+1’ the calculator section displays the result.

- If the command is **neither a color or a sum** then the command is displayed in the unknown section of the application.

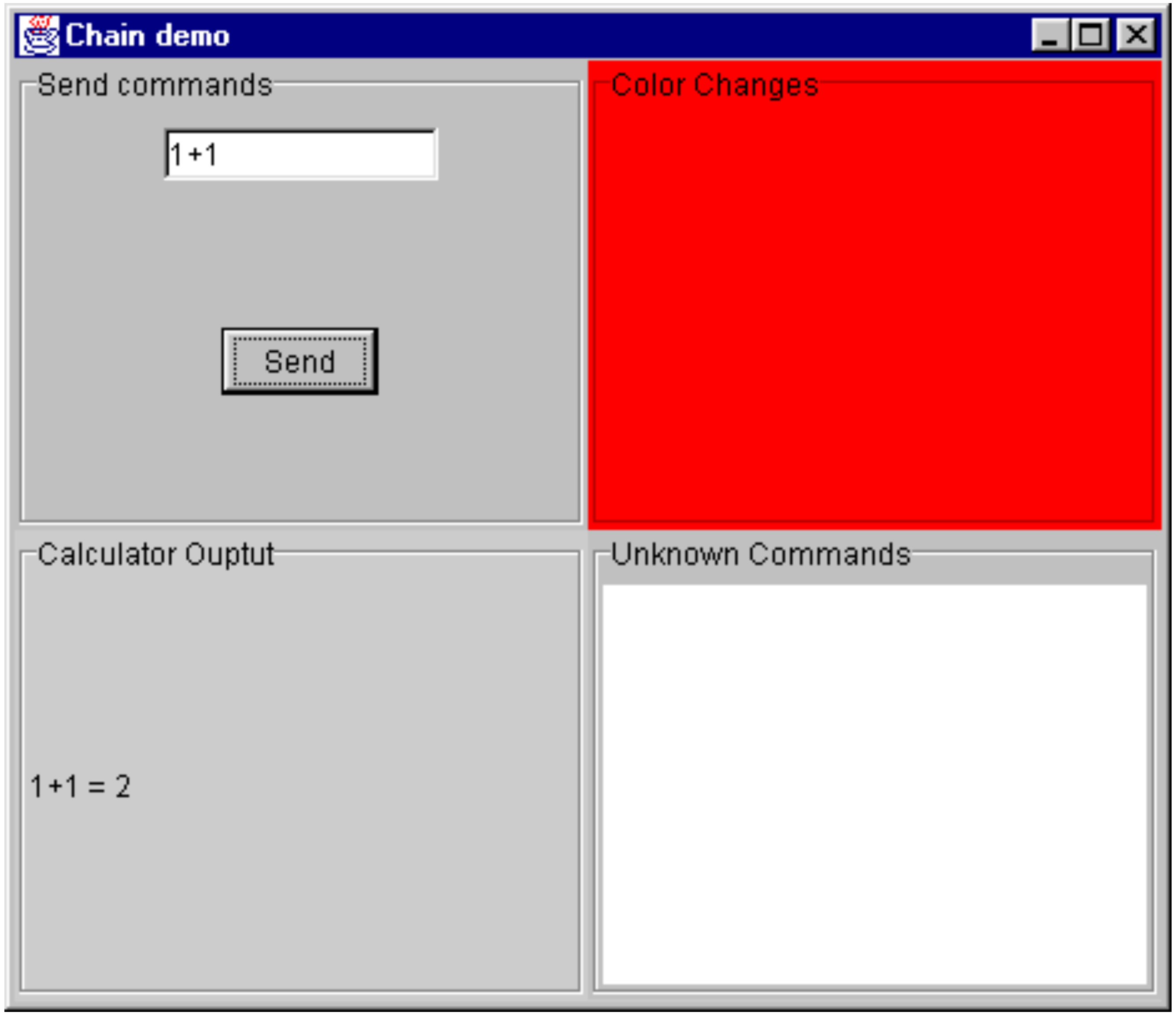
Chain of Responsibility



Chain of Responsibility



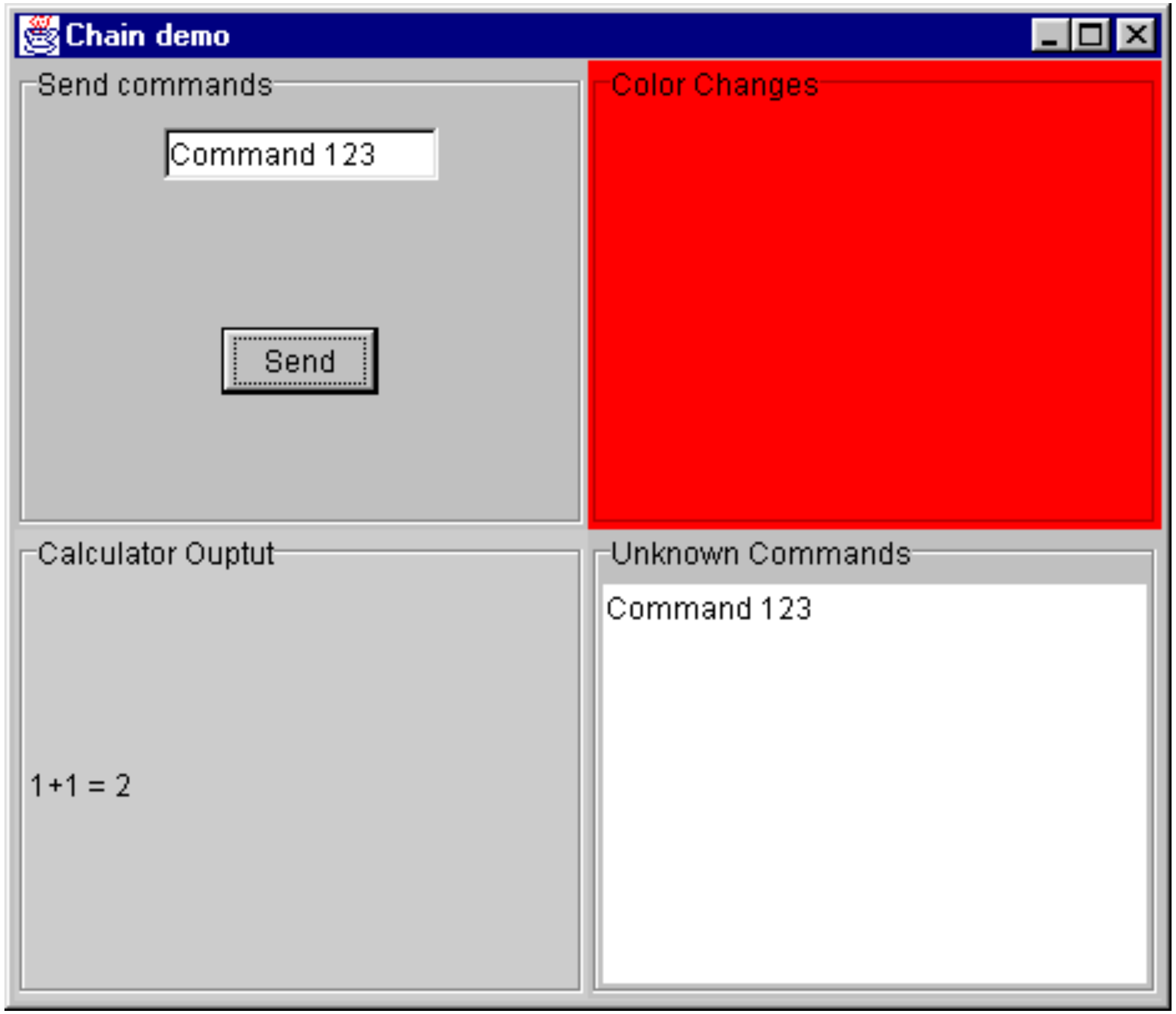
Chain of Responsibility



1+1

1+1 = 2

Chain of Responsibility

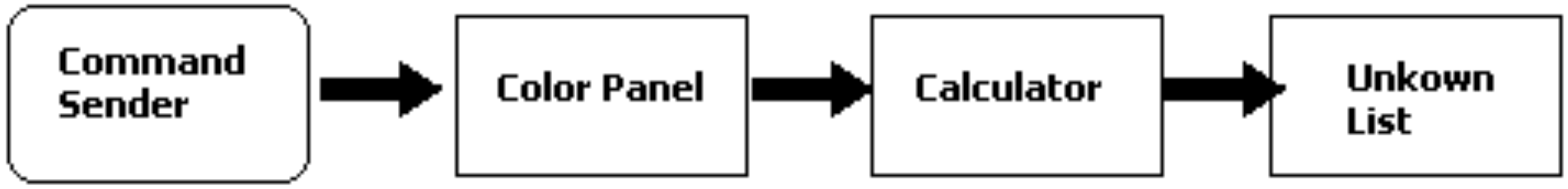


Command 123

1+1 = 2

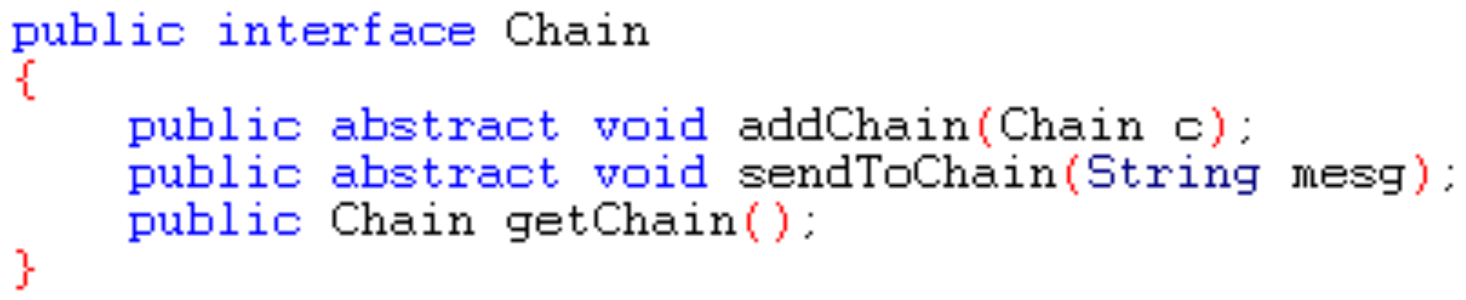
Command 123

Chain of Responsibility



Chain of Responsibility

- The first thing we need to do in order to implement this functionality is to create a Chain interface.



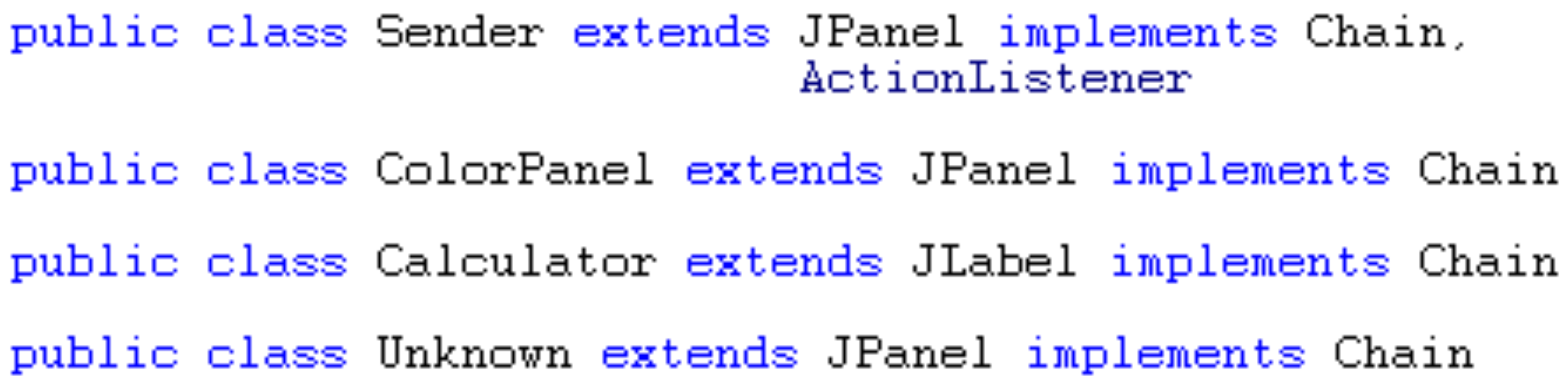
```
public interface Chain
{
    public abstract void addChain(Chain c);
    public abstract void sendToChain(String mesg);
    public Chain getChain();
}
```

- The *addChain* method **adds another class to the chain of classes**.
- **The *getChain* method returns the current class to which messages are being forwarded**. These two methods allow us to modify the chain dynamically and add additional classes in the middle of an existing chain

- The *sendToChain* method **forwards a message to the next object** in the chain.

Chain of Responsibility

- In order for an object to become part of a chain it must implement the Chain interface. This means that our four sections must implement the chain interface.



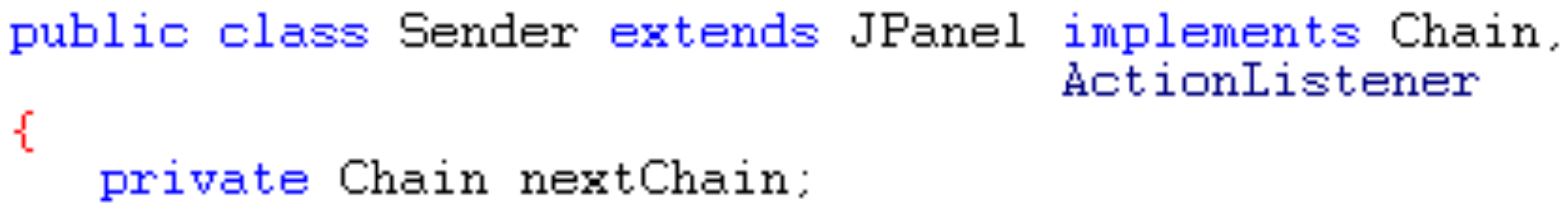
```
public class Sender extends JPanel implements Chain,
                                              ActionListener

public class ColorPanel extends JPanel implements Chain

public class Calculator extends JLabel implements Chain

public class Unknown extends JPanel implements Chain
```

- The most important part of the above classes will be their implementations of the *addChain,getChain* and *sendToChain* methods.

- So lets look at how each class implements the Chain interface.

Sender Chain Class

- The Sender is where the user can enter text and press the send button to forward the message to the next object in the chain. As this object is essentially the first object in the chain its methods are quite simple.

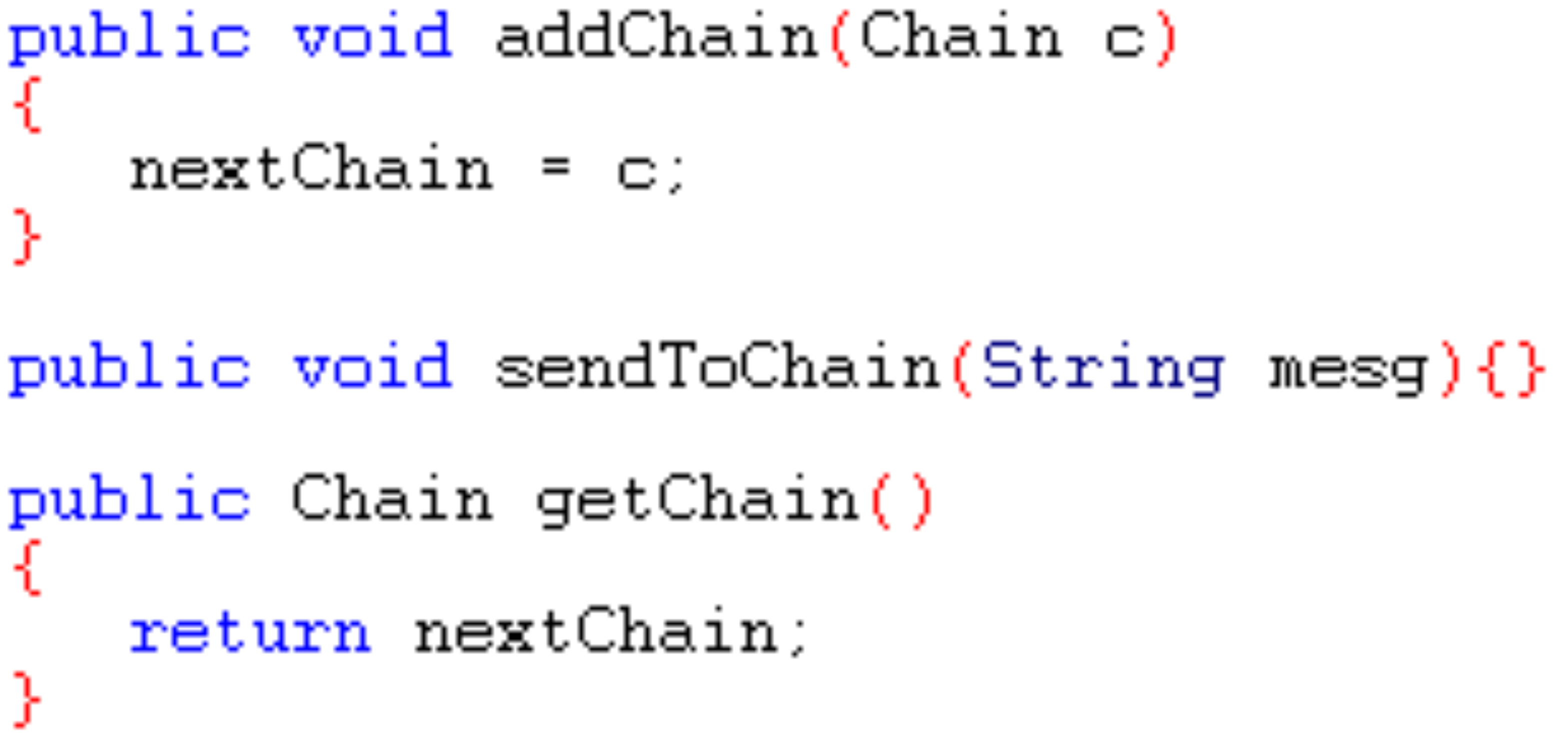


```
public class Sender extends JPanel implements Chain,
                                              ActionListener
{
   private Chain nextChain;
```

- Firstly it must store a reference to the next object in the chain. Its stores this in its nextChain data member. **Every object that is part of a chain will have to store its own reference to the next object**.

Sender Chain Class

- The *addChain,* and *getChain* methods will be the same for all objects in our chain so we will only show them once. <u>Remember that they will have to be implemented in each Chain object.</u>

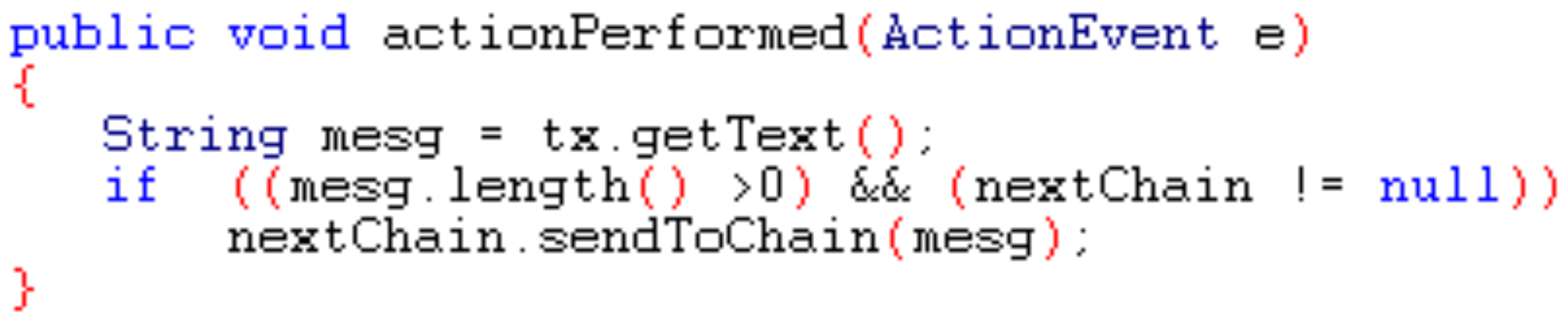


```
public void addChain(Chain c)
{
    nextChain = c;
}

public void sendToChain(String mesg){}

public Chain getChain()
{
    return nextChain;
}
```

Sender Chain Class

- Because the Sender is the first object in the chain its *sendToChain* method can be left blank. This is because it will **never receive a message**, it will only forward messages.

- The sender has an actionPerformed method that simply takes the input from the text field and forwards it to the next object in the chain.



```
public void actionPerformed(ActionEvent e)
{
   String mesg = tx.getText();
   if  ((mesg.length() >0) && (nextChain != null))
       nextChain.sendToChain(mesg);
}
```

ColorPanel Chain Class

- The next object in the chain is the Color Panel. This object also implements the Chain interface. This time the ColorPanel implementation of *sendToChain* is a little more involved because its in the middle of the chain.

```
public void sendToChain(String mesg)
{
      // Try to convert string to
      // color object
      Color c = getColor(mesg);

      if (c != null)
      {
            setBackground(c);
            repaint();
      }
      else
      {
            if (nextChain != null)
                  nextChain.sendToChain(mesg);
      }
}
```

ColorPanel Chain Class

- The ColorPanel checks to see whether the message can be converted to a color. If the message is a color then the background of the panel changes to that color. However if **the message cannot be converted to a color it forwards the message to the next object in the chain.**

- This implementation of *sendToChain* is the one that was called from the Sender class.

- If the message cannot be converted to a color then **variable c will equal null** and the *sendToChain* method of the next object will be called to pass the message on.

ColorPanel Chain Class

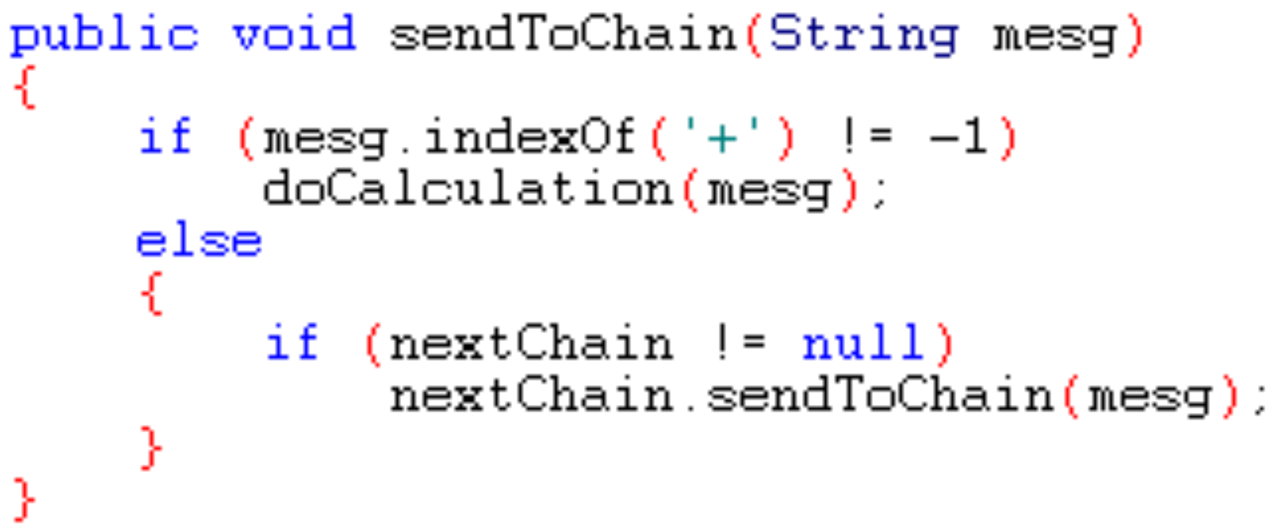
- The color panel uses a simple method to convert the String message to a color.

```
private Color getColor(String mesg)
{
    String lmesg = mesg.toLowerCase();
    Color c = null;

    if (lmesg.equals("red"))
        c = Color.red;
    else if (lmesg.equals("blue"))
        c = Color.blue;
    else if (lmesg.equals("green"))
        c= Color.green;
    return c;
}
```

- **NOTE**: this method is not an integral part of the pattern, its just used to check whether the message is a color message.

Calculator Chain Class

- The next object in the chain is the Calculator. The Calculator’s *sendToChain* method looks as follows:

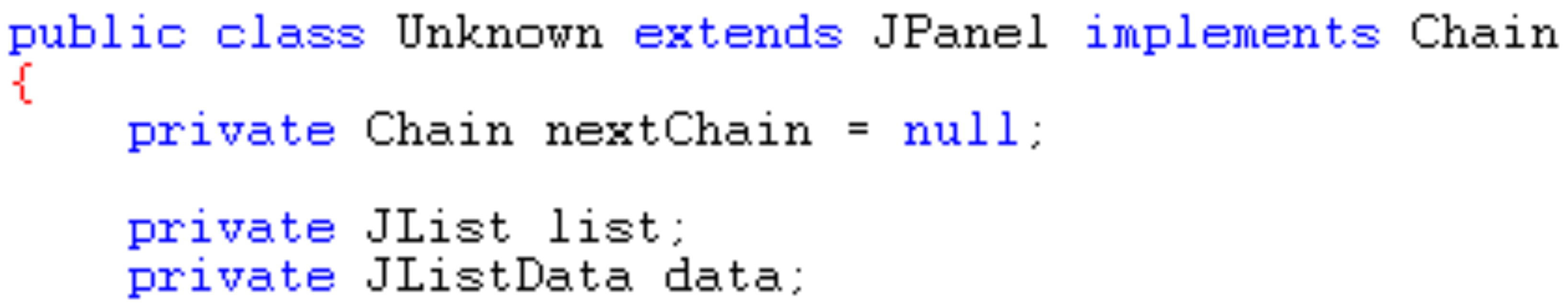


```
public void sendToChain(String mesg)
{
    if (mesg.indexOf('+') != -1)
        doCalculation(mesg);
    else
    {
        if (nextChain != null)
            nextChain.sendToChain(mesg);
    }
}
```

- This simply looks for a ‘+’ symbol in the message and if its finds one it calls the doCalculation method. If there is no ‘+’ symbol in the message then the message is forwarded on to the next object in the chain.

Unknown Chain Class

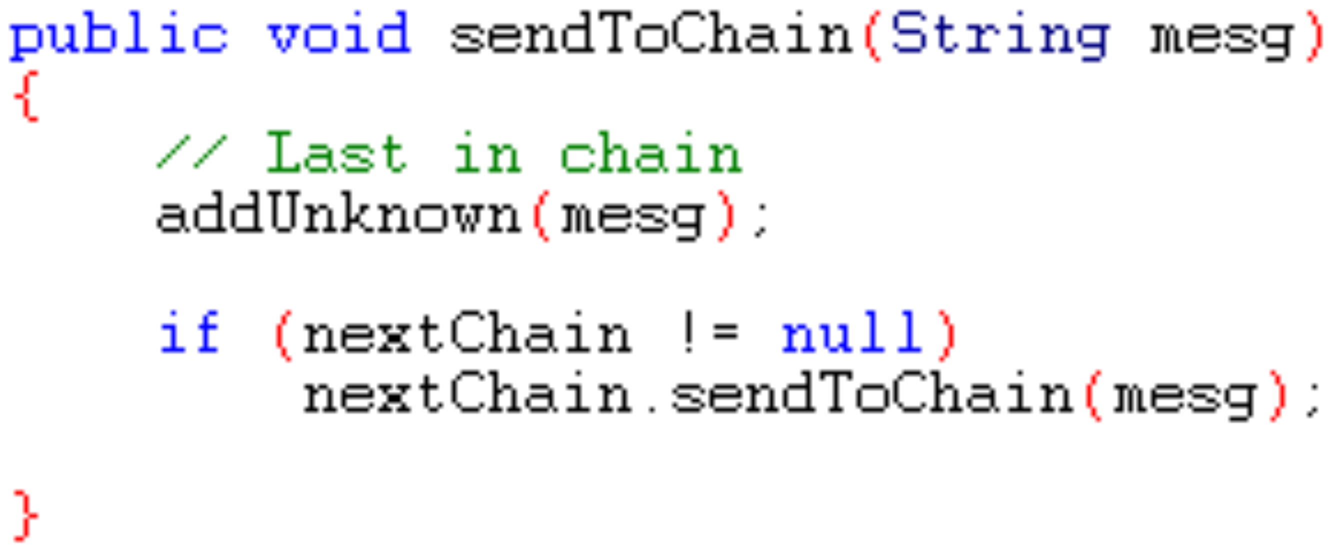
- The final object in the chain is the **Unknown object.** This acts like a **default object** where any **unknown messages are dealt with**. By unknown messages we mean messages that are neither colors or sums. Because this is the last object in the chain its *nextChain* data member is set to null so that it does not try and propagate the message any further.



```
public class Unknown extends JPanel implements Chain
{
    private Chain nextChain = null;

    private JList list;
    private JListData data;
```

Uknown Chain Class

- The *sendToChain* method for the Unknown class looks as follows:



```
public void sendToChain(String mesg)
{
    // Last in chain
    addUnknown(mesg);

    if (nextChain != null)
        nextChain.sendToChain(mesg);

}
```

- The Unknown class uses a JList and an AbstractListModel class called JListData. **When data is added to the JListData object the list is notified and it refreshes itself with the new data.**

Creating the Chain

- The calling program needs to create the Chain objects and connect them together by calling their *addChain* methods. The key parts of the main program are shown below.

- Create the Chain Objects

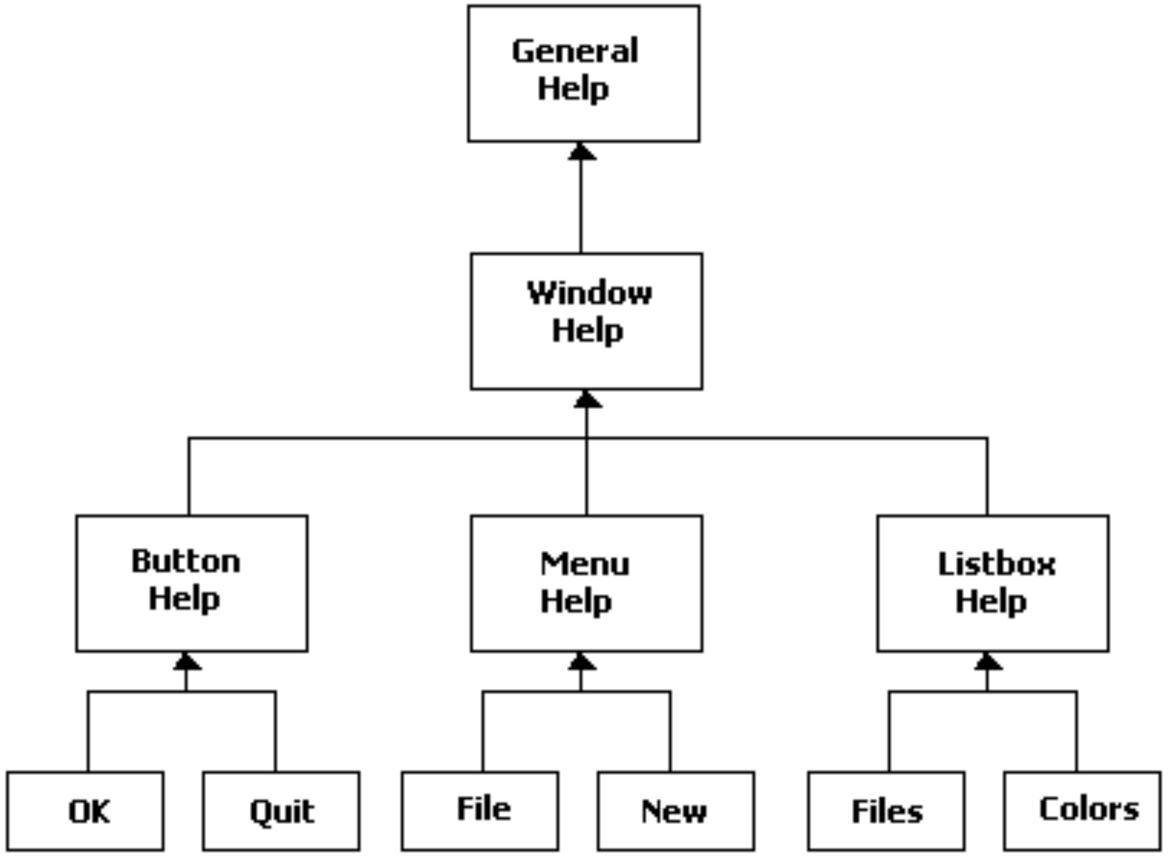
```
sender = new Sender();
colpanel = new ColorPanel();
calc = new Calculator();
unknown = new Unknown();
```

- Connect the Chain Objects

```
sender.addChain(colpanel);
colpanel.addChain(calc);
calc.addChain(unknown);
unknown.addChain(null);
```

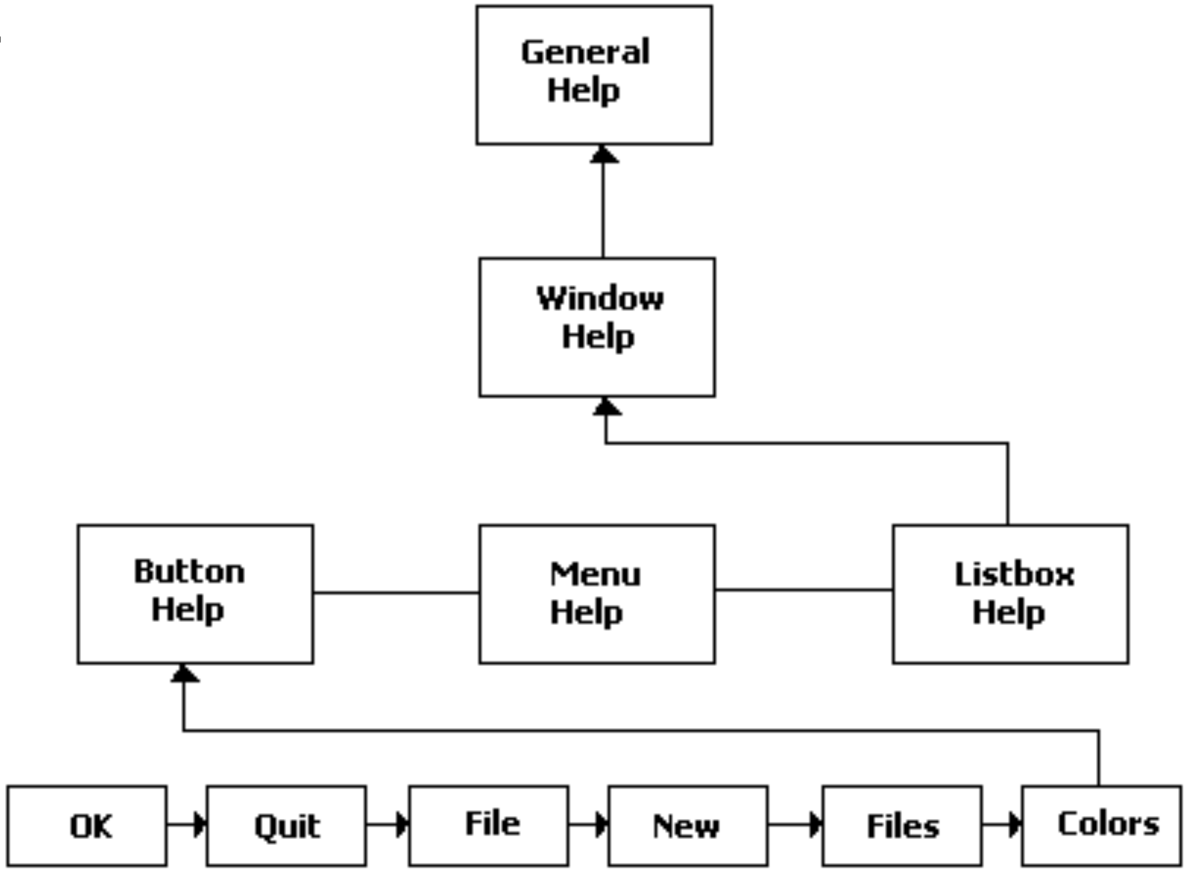
A Chain or Tree

- A Chain of responsibility does not have to be linear. The *Smalltalk Companion* suggests that it is more generally a tree structure with a number of specific entry points (starting points) all pointing upwards to the most general node.



A Chain or Tree

- However this sort of structure seems to imply that **the program will known where to enter the chain**. This can **complicate the design** in some cases. We could instead aline the classes into a single chain, starting at the bottom, and proceeding left to right up one row at a time.



Consequences of the Chain

- This patterns **primary purpose** is to **reduce the coupling between objects**. An object needs only to know how to forward a message to the next object in the chain.
- Each **Java object** in the chain is **self contained**. It knows nothing about the other objects. All each object has to do is **decide** whether **it can satisfy a request**. This makes writing each object and creating the chain very easy.
- You can decide whether you want **the final object** in the chain to **handle all the requests it receives** in some default fashion or it **could just discard them.**

Past Exam Questions

- What is the intent of the **Chain of Responsibility** pattern?
- Describe two consequences of applying the **Chain of Responsibility** pattern to a software design problem
- The **Chain of Responsibility Pattern** helps to keep separate the knowledge of what each object in a program can do. That is it reduces the coupling between objects so that they can act independently.
  Describe four situations where this pattern might be used

Observer Pattern

Observer Pattern

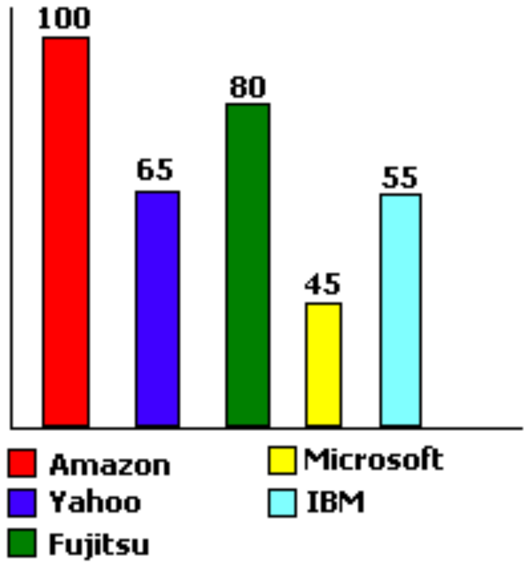
- **Intent:**
  Define a **one-to-one dependancy** between objects so that when **one object changes state**, all **it's dependants** are notified and updated automatically.

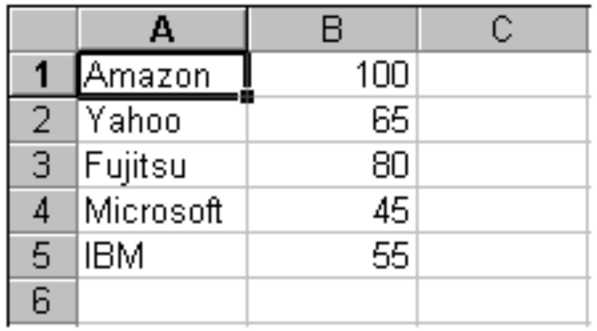
- http://www.codeproject.com/Articles/47948/The-Observer-Design-Pattern

Observer Pattern

- In the more sophisticated windowing world we often **want the data in more than one form at a time** and **have all of the displays reflect any changes in the data.**

- For example, we might represent stock price changes as both a graph and a table or listbox. Each time the price changes, we'd **expect both representations to change at once** without any action on our part.

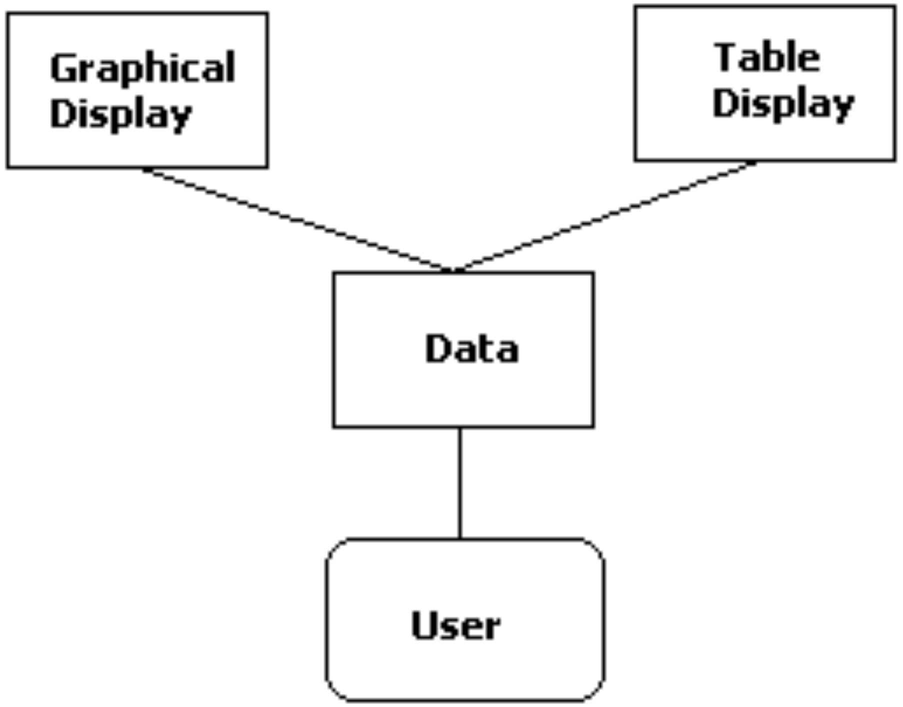




	A	B	C
1	Amazon	100	
2	Yahoo	65	
3	Fujitsu	80	
4	Microsoft	45	
5	IBM	55	
6			

Observer Pattern

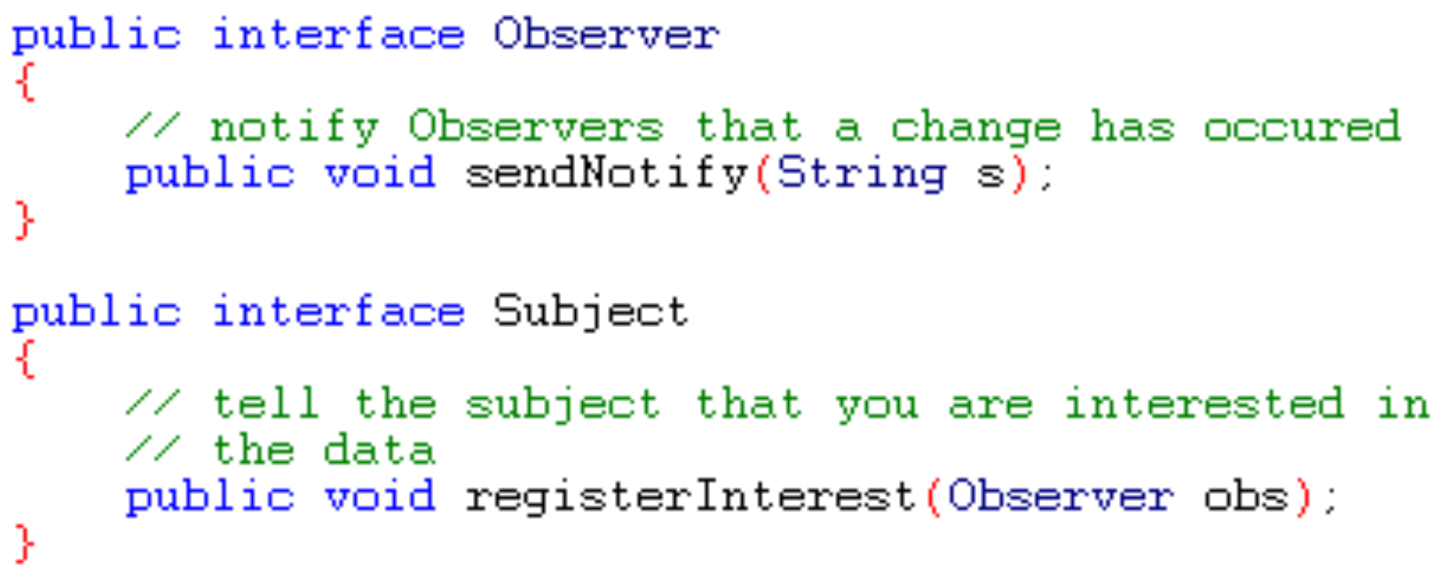
- In Java you can easily use the **Observer pattern** to cause **your program to behave in this way.** The Observer pattern assumes that the object **containing the data** is **separate** from the objects that **display the data** and that these data objects *observe* changes in those data.



Observer Pattern

- When we implement the Observer pattern, we usually refer to the **data as the Subject** and **each of the displays as Observers**. Each Observer registers interest in the data by calling a public method in the Subject, and each display object has a known interface that the subject can call when the data changes.

- We could define these interfaces as follows:



```
public interface Observer
{
    // notify Observers that a change has occured
    public void sendNotify(String s);
}

public interface Subject
{
    // tell the subject that you are interested in
    // the data
    public void registerInterest(Observer obs);
}
```

Observer Pattern

- The **advantage** of defining these abstract interfaces are that you can **write any sort of classes** you want and as long as they **implement these interfaces** they will **automatically refresh their data** when the subject **changes.**

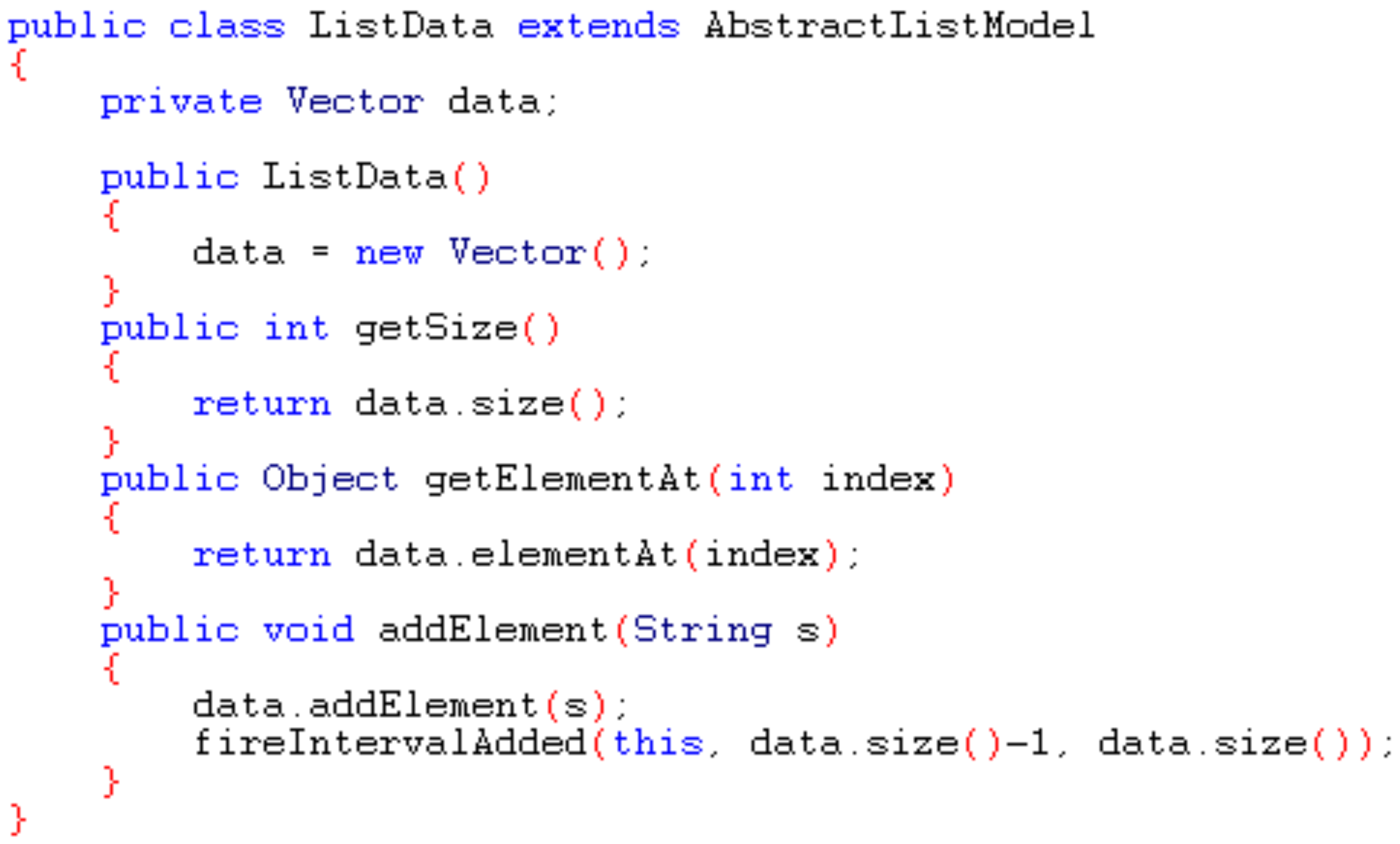
Observer Pattern in Java

- In Java the **JList, JTable, and JTree** objects all operate as **Observers of a data model**. In fact, all of the visual components derived from JComponent can have this same division of labor between the data and the visual representation.

- In **JFC terminology** this is called the **Model View Controller (MVC).**

- In order to take advantage of the MVC architecture in Java we must **first create a Data Model class**. If we are dealing with a list for example we can **extend the *AbstractListModel* class** and create our own data model which we can use in conjunction with a JList control.

Observer Pattern in Java

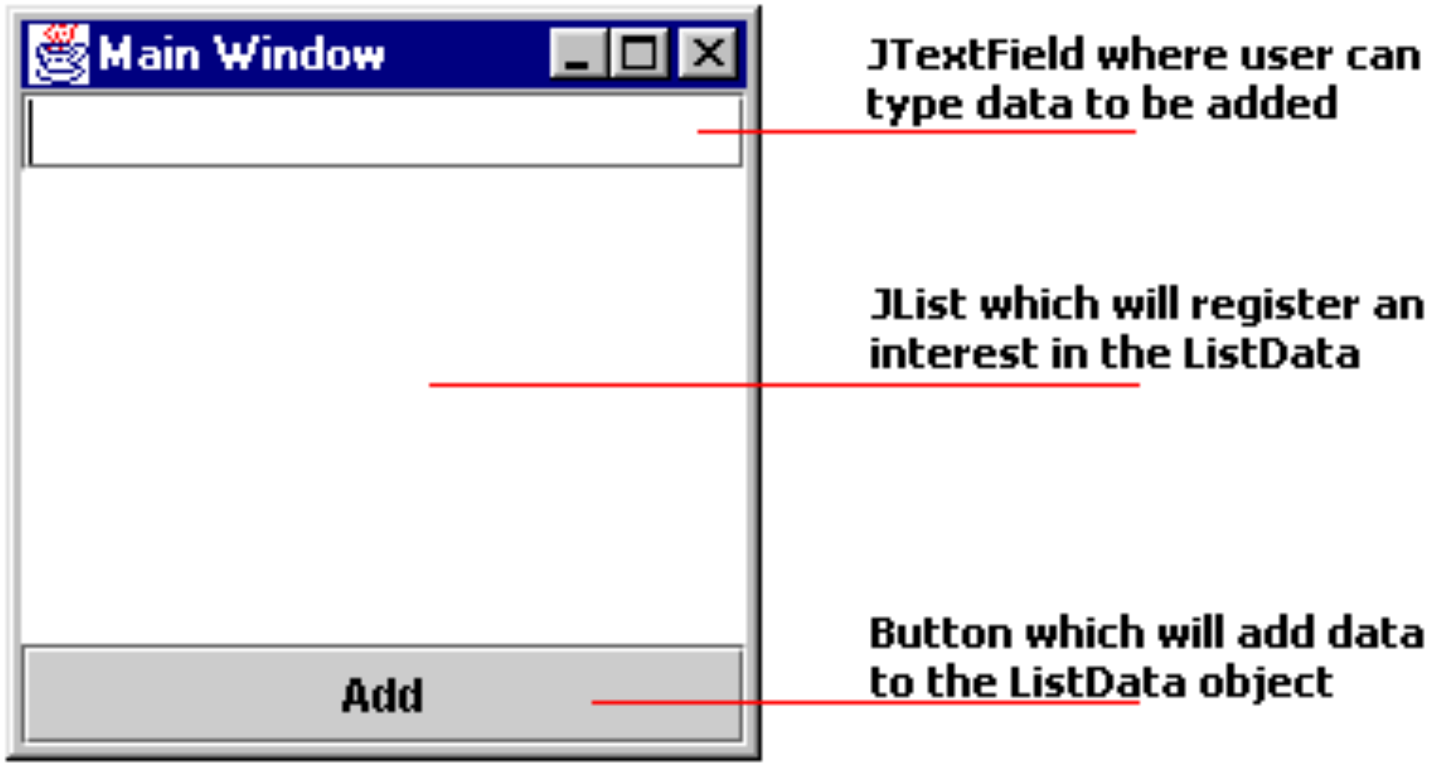


```java
public class ListData extends AbstractListModel
{
    private Vector data;

    public ListData()
    {
        data = new Vector();
    }
    public int getSize()
    {
        return data.size();
    }
    public Object getElementAt(int index)
    {
        return data.elementAt(index);
    }
    public void addElement(String s)
    {
        data.addElement(s);
        fireIntervalAdded(this, data.size()-1, data.size());
    }
}
```

Observer Pattern in Java

- This class **contains a Vector object** which it uses to store data. It also provides methods to gain access to the data and to add new data. This class **represents the Data/Subject part of the Observer pattern.**
- Whenever data is added to the Vector the ***fireIntervalAdded*** method is called. **This method notifies any Observers who have registered an interest in this Subject/Data**.
- There are also equivalent methods for two other kinds of changes in the data; ***fireIntervalRemoved*** and ***fireContentsChanged***.
- The following slide shows how the data model can be used in conjunction with the JList.

Observer Pattern in Java


Main Window
Add
JTextField where user can type data to be added
JList which will register an interest in the ListData
Button which will add data to the ListData object

```
public class MainWindow extends JFrame
                implements ActionListener
{
    private JButton add;
    private JTextField text;
    private JList list;
    private ListData listdata;

    public MainWindow()
    {
        super("Main Window");

        add = new JButton("Add");
        add.addActionListener(this);

        text = new JTextField();
        listdata = new ListData();
        list = new JList(listdata);

        Container c = getContentPane();
        c.setLayout(new BorderLayout());
        c.add(BorderLayout.NORTH, text);
        c.add(BorderLayout.SOUTH, add);
        c.add(BorderLayout.CENTER, list);

        setSize(200,200);
        show();
    }
```

Observer

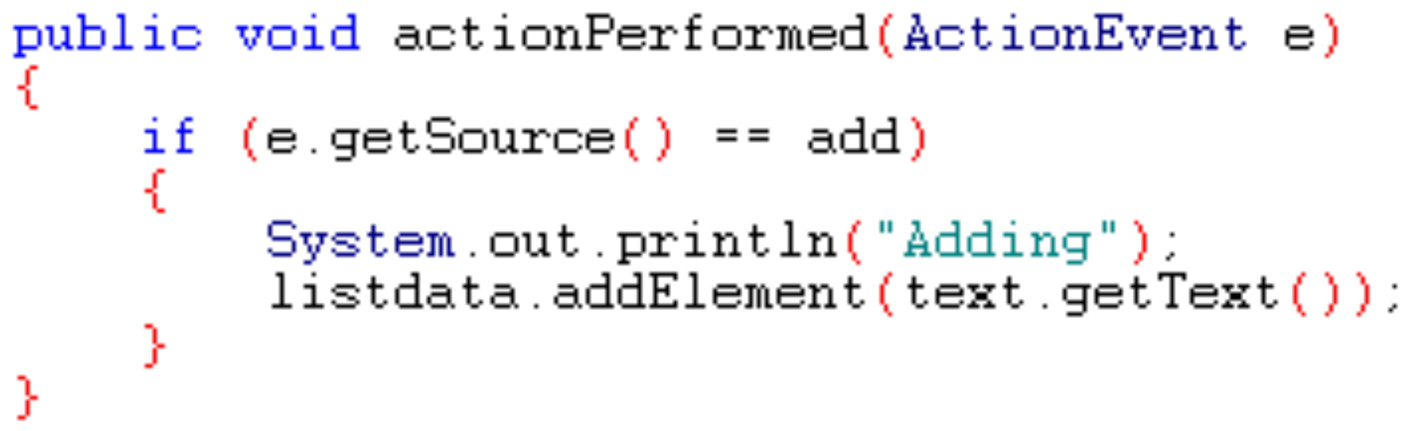
Data Model

Register Interest in Data

Observer Pattern in Java

- When we **create the JList object** we **pass the ListData object as a parameter to the constructor.** This effectively **tells the ListData** that **the JList has registered an interest in its data**.

- The **application can then add data to the ListData object** and the **JList will be automatically updated.**



```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == add)
    {
        System.out.println("Adding");
        listdata.addElement(text.getText());
    }
}
```

Consequences of the Observer

- The observer pattern promotes abtract coupling to Subjects.
  A Subject doesn't know the details of any of its Observers.
  This has the potential **disadvantage** of **successive or repeated updates to the Observers** when there is series of incremental changes to the data.

- If the **cost** of these updates is **high**, introducing some sort of **change management** might be necessary so that the Observers are not notified too soon to often.

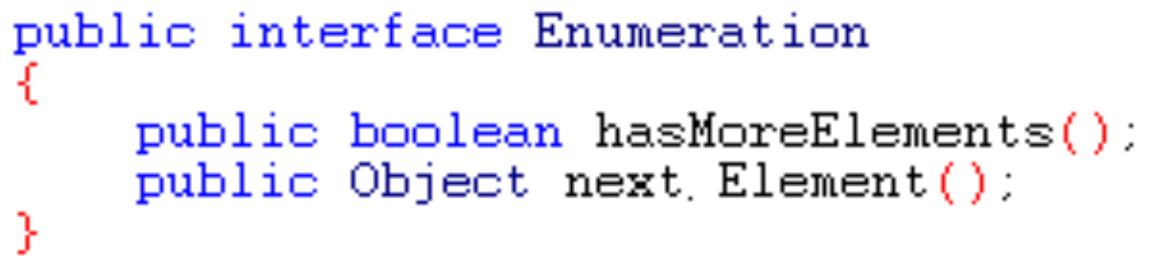
Iterator Pattern

Iterator

- The Iterator pattern is one of the **simplest** and **most frequently** used design patterns. It allows you to move through a list of data using a **standard interface** without having **to know the details of that data's internal representation**.
- You can also **define some special iterators** that perform some special processing and return only specified elements of the data collection.
- The Iterator pattern is useful because it provides **a defined way to move through a set of data elements without exposing what is taking place inside the class.**
- It **is an *interface***; thus you can implement it in any way that is convenient for the data that you are returning.

Iterator

- Java has its own Iterator interface which is used extensively with the Vector and HashTable classes.



```
public interface Enumeration
{
    public boolean hasMoreElements();
    public Object next. Element();
}
```

- Not having a method to move to the top of a list might seem restrictive at first. However it is not a serious problem in Java because you customarily obtain a new instance of the Enumeration each time you want to move through a list.

Enumerations in Java

- The Enumeration type is built into the Vector and HashTable classes. Rather then these classes implementing the two methods of the Enumeration directly, both contain an *elements* method that returns an Enumeration of that class's data.

```
public Enumeration elements();
```

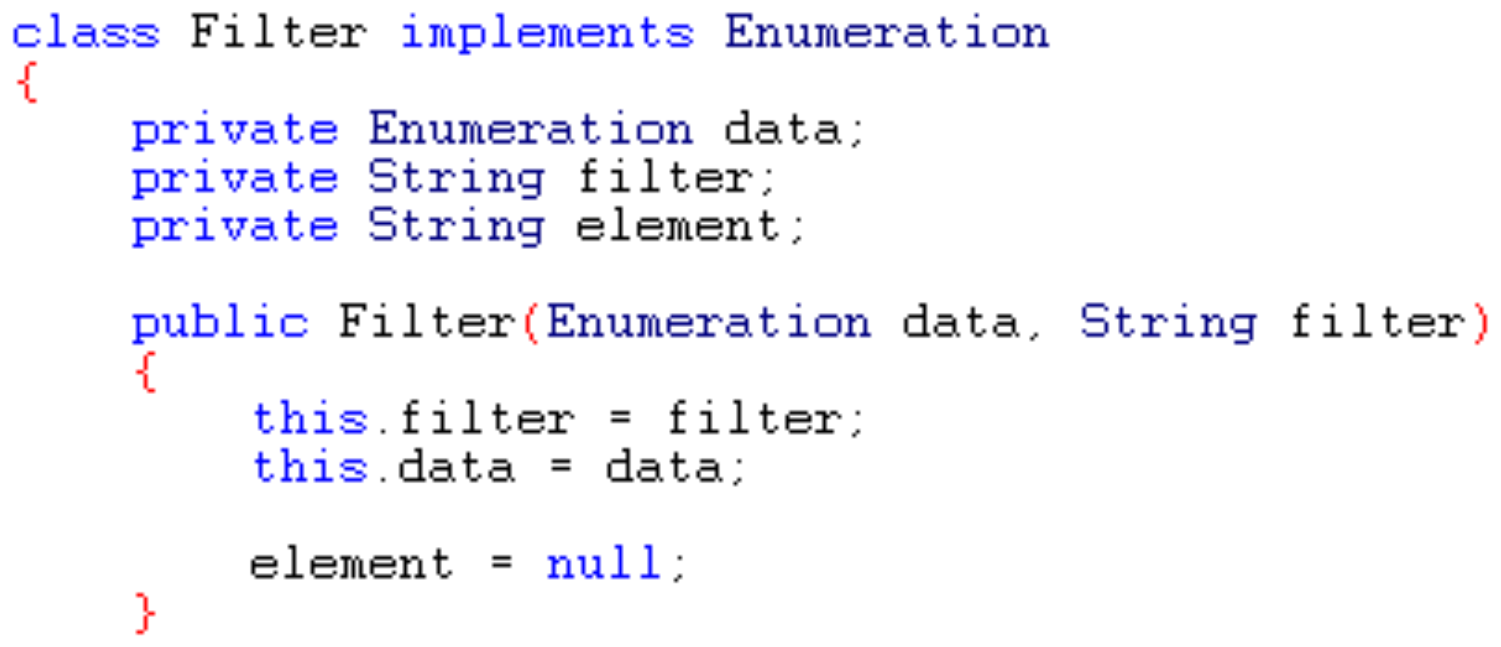
- This ***elements* method** is really a kind of **Factory method** that produces instances of an Enumeration class. You can then move through the list using the following code:

```
Enumeration e = vector.elements();
while(e.hasMoreElements())
{
    String name = (String)e.nextElement();
    System.out.println(s);
}
```

Filtered Iterators

- Having a clearly defined method of moving through a collection is helpful. However, you can also define **filtered Enumerations** that **perform some computation on the data before returning it.**
- For example, **you could return the data in some particular order or return only those objects that match a certain criteria.**
- Suppose that we want to enumerate only those names in a list that start with the letter ‘J’. In other words we would like to iterate through a list ignoring every name that does not start with ‘J’.
- The first step is to write a class that implements the Enumeration interface.

Filtered Iterators

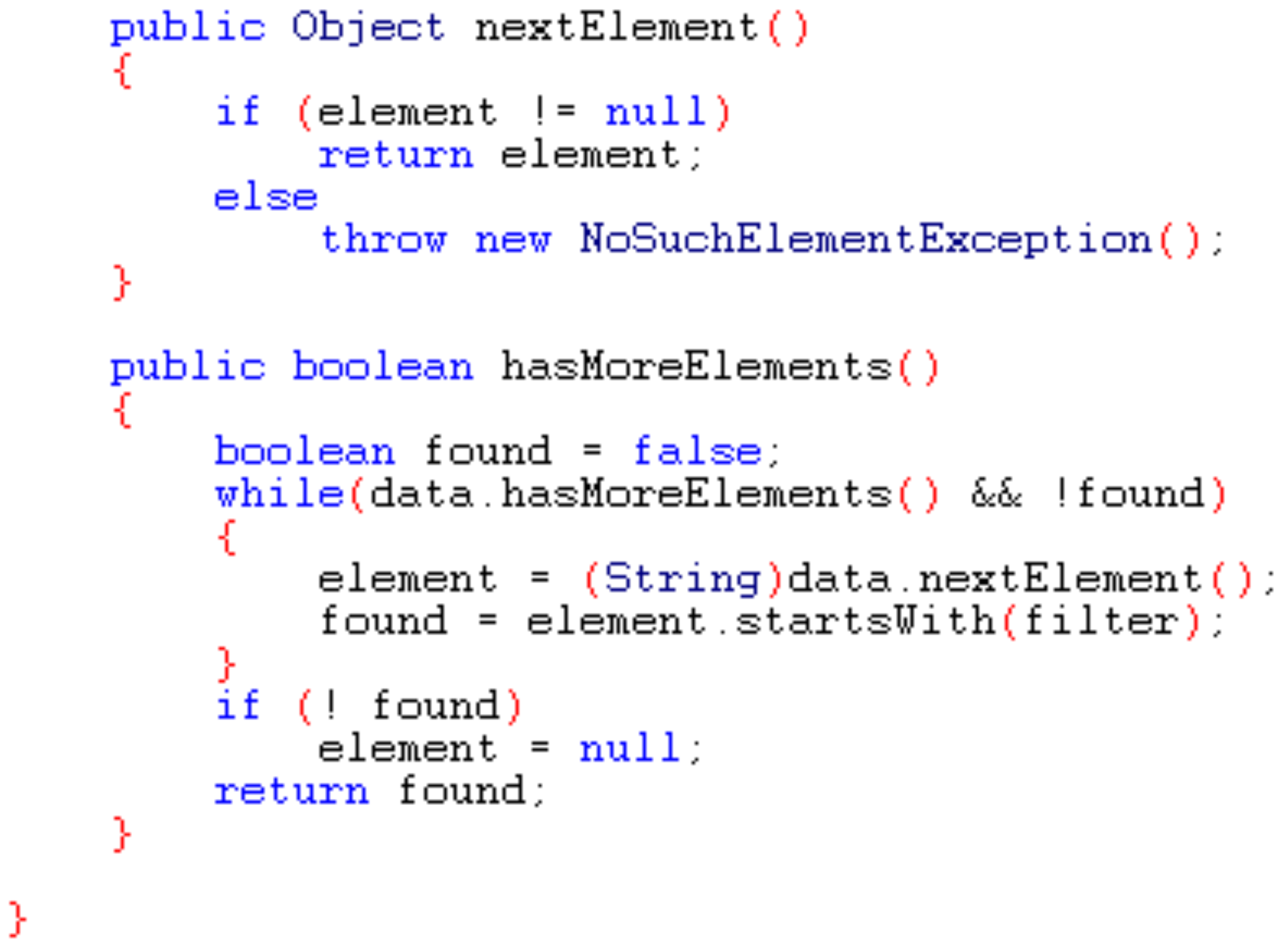


```
class Filter implements Enumeration
{
    private Enumeration data;
    private String filter;
    private String element;

    public Filter(Enumeration data, String filter)
    {
        this.filter = filter;
        this.data = data;

        element = null;
    }
```

- The Filter class implements the Enumeration interface. Its constructor takes data from another Enumeration and also takes a String to check the data against. The Filter class also stores the current element in its element data member.

Filtered Iterators

- The Filter class must also provide implementations for the *hasMoreElements* and *nextElement* methods.



```
    public Object nextElement()
    {
        if (element != null)
            return element;
        else
            throw new NoSuchElementException();
    }

    public boolean hasMoreElements()
    {
        boolean found = false;
        while(data.hasMoreElements() && !found)
        {
            element = (String)data.nextElement();
            found = element.startsWith(filter);
        }
        if (! found)
            element = null;
        return found;
    }
}
```

Filtered Iterators

- All of the work is done in the *hasMoreElements* method. This method scans through the collection for an Object (in this case String) that starts with a given letter. It either **saves the String in the *element* variable** or **sets it to null and returns either *true* or *false*.**

- The *nextElement* method either returns that next *element* or throws an exception if there are no more elements in the collection.

- All we need now is a test program and a Vector so that we can use the Filter class.

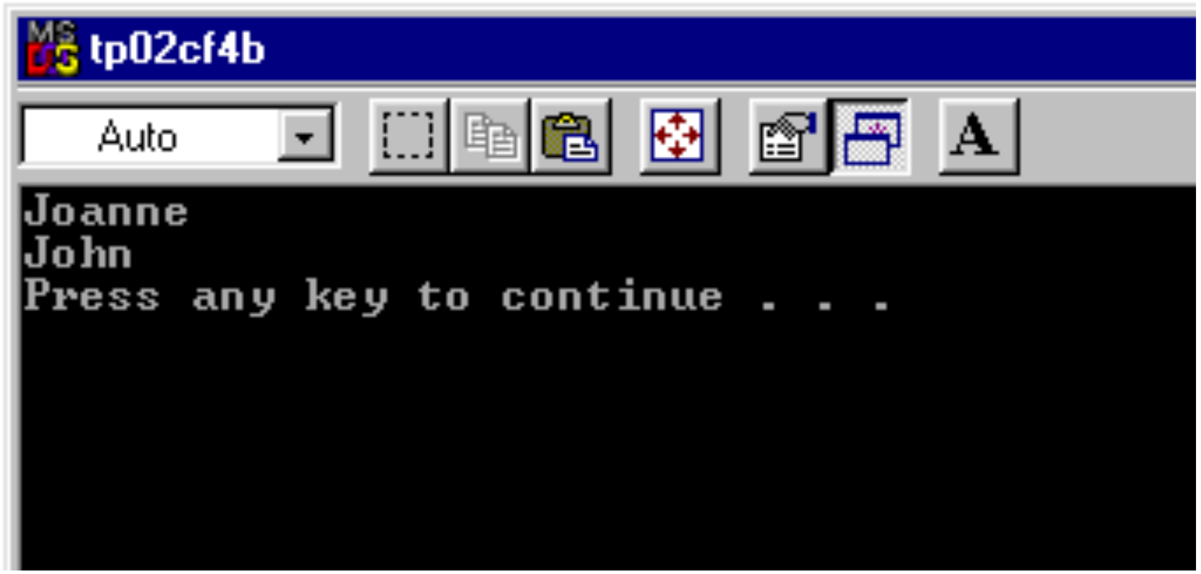
- The test program is shown on the next slide.

Filtered Iterators

```
class MainApp
{
    private Vector data;

    public MainApp()
    {
        data = new Vector();

        data.addElement("Alan");
        data.addElement("Eimear");
        data.addElement("Conor");
        data.addElement("Andrew");
        data.addElement("Joanne");
        data.addElement("David");
        data.addElement("John");
        data.addElement("Martin");
    }

    public void filterNames()
    {
        Filter filter = new Filter(data.elements(), "J");

        while(filter.hasMoreElements())
        {
            String s = (String)filter.nextElement();
            System.out.println(s);
        }
    }
```

Filtered Iterators

```
        public static void main(String[] args)
        {
            MainApp app = new MainApp();

            app.filterNames();
        }
    }
```



```
Joanne
John
Press any key to continue . . .
```

Iterator Consequences

- **Data Modification**
  - The most significant question resulting from the use of the iterator **concerns iterating through data while it is being changed**. There are no simple solutions to this problem. You could make the *nextElement* method thread safe by making it ***synchronized*** which would guarantee that the loop would finish before any other code got a chance to change the contents of the collection.

- **Privileged Access**
  - **Enumeration classes** might have to have **privileged access** to the **underlying data** of the original container so that they can **move through the data**. In the case of a Vector or HashTable this is easy but if it's an underlying data structure that you have created you will have to provide access to it.