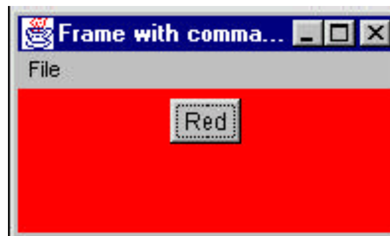
Command Pattern

What is it?

- The Chain of Responsibility forwards requests along a chain of classes, but the Command pattern forwards a request only to a specific module.
- It encloses a request for a specific action inside an object and gives it a known public interface.
- It lets you give the client the ability to make requests without knowing anything about the actual action that will be performed, and allows you to change that action without affecting the client program in any way.

For Example

- When you build a Java user interface, you provide menu items, buttons, and checkboxes and so forth to allow the user to tell the program what to do.
- When a user selects one of these controls, the program receives an *ActionEvent*, which it must trap by subclassing, the *actionPerformed* event.
- Let's suppose we build a very simple program that allows you to select the menu items File | Open and File | Exit, and click on a button marked Red which turns the background of the window red.
- The program consists of the File Menu object with the mnuOpen and mnuExit MenuItems added to it.
- It also contains one button called btnRed.
- A click on any of these causes an *actionPerformed* event that we can trap.



The actionPerformed() Code

```
public void actionPerformed(ActionEvent e)    {
      Object obj = e.getSource();
      if(obj == mnuOpen)
         fileOpen();              //open file
      if (obj == mnuExit)
         exitClicked();           //exit from program
      if (obj == btnRed)
         redClicked();            //turn red
  }
```

The Methods Code

- The three private methods this method calls are…

```
private void exitClicked()    {
   System.exit(0);
}
//----------------------------------------
private void fileOpen()     {
 FileDialog fDlg = new FileDialog(this, "Open a file",
                             FileDialog.LOAD);
   fDlg.show();
}
//----------------------------------------
private void redClicked()     {
   p.setBackground(Color.red);
}
```


Good for Small Examples but…

- This approach works fine as long as there are only a few menu items and buttons, but
  - when you have dozens of menu items and several buttons, the *actionPerformed* code can get pretty unwieldy.
- In addition, this really seems a little inelegant, since we'd really hope that in an object-oriented language like Java, we could avoid a long series of if statements to identify the selected object.
- Instead, we'd like to find a way to have each object receive its commands directly.

Our Goal…

- The objective is to reduce the actionPerformed method to:

```
public void actionPerformed(ActionEvent e)  {
       Command cmd = (Command)e.getSource();
       cmd.Execute();
   }
```


The Command Object

- One way to assure that every object receives its own commands directly is to use the Command object approach.
- A Command object always has an Execute() method that is called when an action occurs on that object.
- Most simply, a Command object implements at least the following interface.

```
public interface Command  {
   public void Execute();
}
```

The Command Philosophy

- Provide an Execute method for each object which carries out the desired action, thus keeping the knowledge of what to do inside the object where it belongs, instead of having another part of the program make these decisions.
- One important purpose of the Command pattern is to keep the program and user interface objects completely separate from the actions that they initiate.
  - In other words, these program objects should be completely separate from each other and should not have to know how other objects work.
- The user interface receives a command and tells a Command object to carry out whatever duties it has been instructed to do.
  - The UI does not and should not need to know what tasks will be executed.
- The Command object can also be used when you need to tell the program to execute the command when the resources are available rather than immediately.
  - In such cases, you are queuing commands to be executed later.
- Finally, you can use Command objects to remember operations so that you can support Undo requests.

Approach

- Derive new classes from the MenuItem and Button classes and implement the Command interface in each.

```
class btnRedCommand extends Button
                    implements Command     {
      public btnRedCommand(String caption)       {
         super(caption);   //initialize the button
      }
      public void Execute()      {
         p.setBackground(Color.red);
      }
   }
   //-------------------------------------------
class fileExitCommand extends MenuItem
                    implements Command    {
      public fileExitCommand(String caption)       {
         super(caption);         //initialize the Menu
      }
      public void Execute()       {
         System.exit(0);
      }
   }
```

Advantages/Disadvantages

- This lets us simplify the calls made in the actionPerformed method.
- But it requires that we create and instantiate a new class for each action we want to execute.

```
mnuOpen.addActionListener(new fileOpen());
mnuExit.addActionListener(new fileExit());
btnRed.addActionListener(new btnRed());
```