MVC

- Structuring software
- Writing reusable code
- Flexibility
- Model-View-Controller (MVC)

Software Structure

- Even in the simplest application there are many different ways of structuring the code that runs
- How the code is split between the various:
  - Classes
  - Methods
  - Variables
- And how these components interact with each other

Software Structure

- These decisions are all made by the developer (you)
- The structure of the software is unknown to the end-user and they will not know how the code was structured
- The same is true of the computer. The Java compiler does not care how you structure the code
- The code is structured purely for the benefit of the developer

Software Structure

- Neither the compiler or the end user care if you write a program that:
  - Is one 9000 line method
  - Or a thousand 9 line methods spread among hundreds of classes

Software Structure

- Breaking code into smaller chunks (Classes and methods) has two main advantages
  - Reducing repeated code
  - Making it easier to replace sections of the code, increasing flexibility

Reducing repeated code

- At its simplest level, breaking software into smaller pieces can reduce repetition. For example, form Week 1:

```java
public static void drawShape() {
    System.out.println("+----+");
    System.out.println("|    |");
    System.out.println("|    |");
    System.out.println("+----+");
}

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

```java
public static void main(String[] args) {
    System.out.println("+----+");
    System.out.println("|    |");
    System.out.println("|    |");
    System.out.println("+----+");

    System.out.println("+----+");
    System.out.println("|    |");
    System.out.println("|    |");
    System.out.println("+----+");
}
```

Repeated code

- Repeated code is bad because it is difficult to make changes to. If you want to change the code you have to change it in multiple places

```java
public static void drawShape() {
        System.out.println("  /\\ ");
        System.out.println(" /  \\ ");
        System.out.println("/    \\");
        System.out.println("+----+");

}

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

```java
public static void main(String[] args) {
        System.out.println("  /\\ ");
        System.out.println(" /  \\ ");
        System.out.println("/    \\");
        System.out.println("+----+");

        System.out.println("  /\\ ");
        System.out.println(" /  \\ ");
        System.out.println("/    \\");
        System.out.println("+----+");
}
```

Repeated code

- Less code can sometimes take longer to write even though you are writing less
- Thinking about how to structure the software takes time
- However, the result is usually worthwhile as further developments are faster

Flexibility

- The second reason for separating code into methods and classes is flexibility
- A lot of the time you will have sections of the code that do something *similar* but not *identical*
- For example, you may want to draw a shape on the screen but perhaps there is one place you want to draw a triangle and another a square

Flexibility

- By separating these out into classes they become more reusable. It's possible to write a method that can take any shape
- If a method can work with different classes it's infinitely more flexible.

Flexibility

- It's possible to write a long method that contains all the relevant parts:

```java
public static void draw(String shape, int times) {
    for (int i = 0; i < times; i++) {
        if (shape.equals("square")) {
            System.out.println("+----+");
            System.out.println("|    |");
            System.out.println("|    |");
            System.out.println("+----+");
        }
        else if (shape.equals("triangle")) {
            System.out.println("  /\\ ");
            System.out.println(" /  \\ ");
            System.out.println("/    \\");
            System.out.println("+----+");
        }
        else if (shape.equals("diamond")) {
            System.out.println("  /\\ ");
            System.out.println(" /  \\ ");
            System.out.println("\\  /  ");
            System.out.println(" \\/   ");
        }
    }
}
```

```java
//Draw the triangle 4 times
draw("triangle", 4);
//Draw the diamond twice
draw("diamond", 2);
//Draw the square once
draw("square", 1);
```

Flexibility

- This code works but it's needlessly long and inflexible
- To add another shape I have to amend the method and add an extra if statement
- If there are 20 shapes this gets very long and difficult to manage
- It's also inefficient The if statement needs to run for each possible shape

Flexibility

- Writing code that doesn't know about the actual shape in use is preferable. To do this, you can create an interface with a relevant method:

```
public interface Shape {
    public String draw();
    public String getName();
}
```

- Then re-write the original method to use the interface

Flexibility

- It's possible to write a long method that contains all the relevant parts:

```java
public static void draw(String shape, int times) {
    for (int i = 0; i < times; i++) {
        if (shape.equals("square")) {
            System.out.println("+----+");
            System.out.println("|    |");
            System.out.println("|    |");
            System.out.println("+----+");
        }
        else if (shape.equals("triangle")) {
            System.out.println("  /\\ ");
            System.out.println(" /  \\ ");
            System.out.println("/    \\");
            System.out.println("+----+");
        }
        else if (shape.equals("diamond")) {
            System.out.println("  /\\ ");
            System.out.println(" /  \\ ");
            System.out.println("\\  /  ");
            System.out.println(" \\/   ");
        }
    }
}
```

```java
public interface Shape {
    public void draw();
}
```

```java
public class Triangle implements Shape {
    public void draw() {
        System.out.println("  /\\ ");
        System.out.println(" /  \\ ");
        System.out.println("/    \\");
        System.out.println("+----+");
    }
}
```

```java
public static void draw(Shape shape,
                        int times) {
    for (int i = 0; i < times; i++) {
        shape.draw();
    }
}
```

Flexibility

- This reduces the code in draw method and separates it out

```
public static void draw(Shape shape, int times) {
    for (int i = 0; i < times; i++) {
        shape.draw();
    }
}
```

- This makes the code more flexible: It can work with *any* possible shape and will never need to be changed
- Someone can easily extend the code to write any shape by writing the relevant class

Flexibility

- However, it's possible to take this a step further
- It's worth considering concepts in a much more general sense
- Shapes may not be the only think which can be drawn to the screen
- Text, paragraphs, logos, images or almost anything else could be drawn

Flexibility

- This method only needs to be provided an object with a *draw* method yet it can only work with shapes

```
public static void draw(Shape shape, int times) {
    for (int i = 0; i < times; i++) {
        shape.draw();
    }
}
```

Flexibility

- Instead of a shape interface which has a draw method, a more simplified Drawable interface could be created which has a draw method
- This could then be used by any object, shape or non-shape

```java
public interface Drawable {
    public void draw();
}
```

```java
public static void draw(Drawable shape, int times) {
    for (int i = 0; i < times; i++) {
        shape.draw();
    }
}
```

```java
public class E implements Drawable {
    public void draw() {
        System.out.println("EEEEEEE ");
        System.out.println("E       ");
        System.out.println("EEE     ");
        System.out.println("E       ");
        System.out.println("EEEEEEE ");
    }
}
```

```java
public class Player implements Drawable {
    public void draw() {
        System.out.println("   O    ");
        System.out.println("  /|\\   ");
        System.out.println("   |    ");
        System.out.println("  /\\   ");
    }
}
```

Reusability

- By thinking more about how things are split up and making methods and classes as minimal and generic as possible they are more reusable
- Programming around interfaces instead of classes offers much greater flexibility than just using classes

Separation of concerns

- The way software is made reusable is via 'separation of concerns'
- A concern is a single set of related operations. For example:
  - Creating a GUI is a concern
  - Processing the data is a concern
  - Loading/Saving the data from a file is a concern
  - Handling user input is a concern
  - Validating user input is a concern

Separation of Concerns - HTML

- On the web Separation of Concerns is handled by different technologies
  - **HTML** stores the structure and content of the data
  - **CSS** defines how the HTML will be presented
  - **JavaScript** deals with user input (and other) events

Separation of Concerns - HTML

- Because each concern is stored separately in its own file it's possible to reuse any of the components:
  - Style a page completely differently by substituting the stylesheet
  - Use the same stylesheet to style multiple HTML files
  - Use JavaScript to enhance functionality of multiple pages

Separation of Concerns - Java

- In Java the concerns are less obvious because they are all the same language however we can split them up in a similar manner:
  - The user input (Action listeners)
  - Storing the application's data
  - Displaying the GUI

Model View Controller

- There is a common software architecture called Model-View-Controller which defines this separation of concerns
- MVC defines a software component to handle each concern
- As well as defining how the three components are related

Model – Application data

- The Model deals with storing and retrieving the data that the application requires
- Models do all the processing on the data:
  - Text formatting
  - Calculations
  - Storing in files/databases
- **Models do not know about controllers or views**
- **Multiple controllers and views can be using the same model**

View – The GUI

- A View is the *Visible* part of the application. In most cases, the GUI
- Each view stores a specific part of the GUI. This is may be a window, panel or even a single component
- **The view uses a controller for its ActionListener**
- **The view gets its data directly from the model**
- **The view is aware of both the controller and the model**

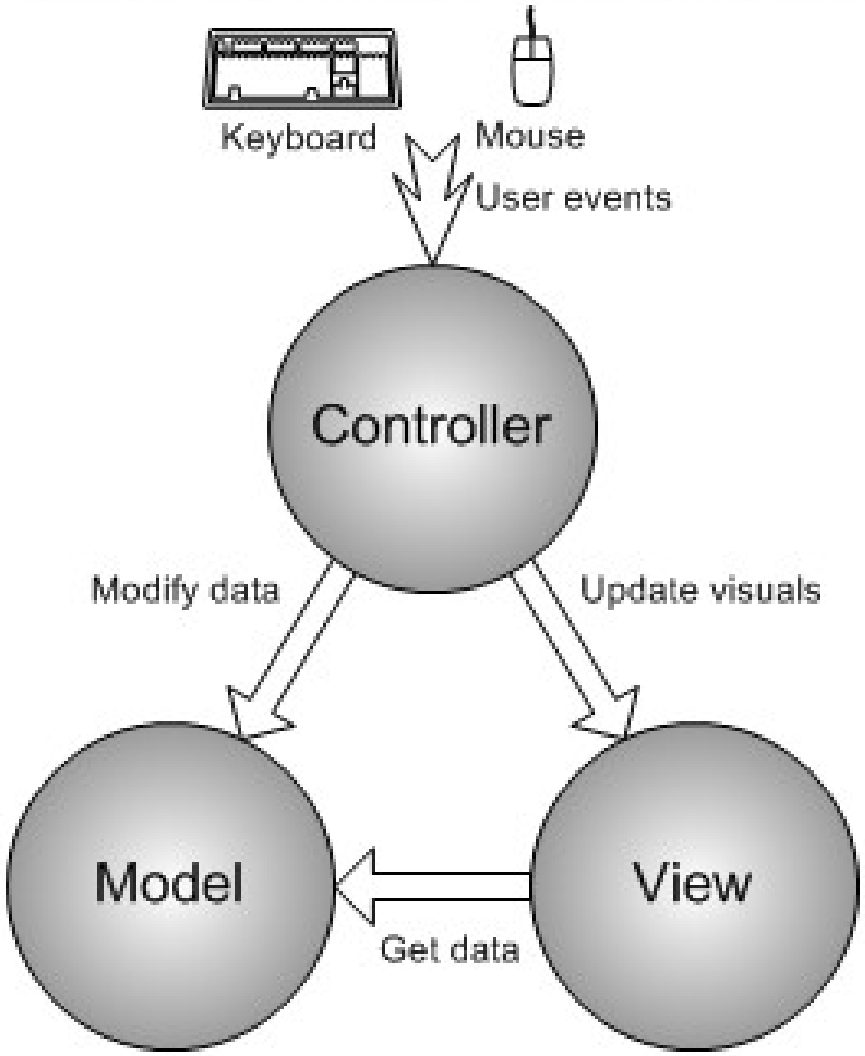
Controllers – User input

- Controllers handle all events within the application
- They usually implement ActionListeners or other GUI event listeners (They can implement more than one Event Listener)
- Controllers know about both the Model and the View
- Controllers can be linked to more than

MVC – Program Flow

- The program flow in MVC is:
  - The View (GUI) is displayed
  - The user interacts with the GUI e.g. clicking a button
  - The Controller (Action Listener) is triggered with some information about what happened in the view (e.g. which button was pressed)
  - The controller updates the model in some way
  - The view is refreshed, reading the updated data from the model

MVC – Program Flow

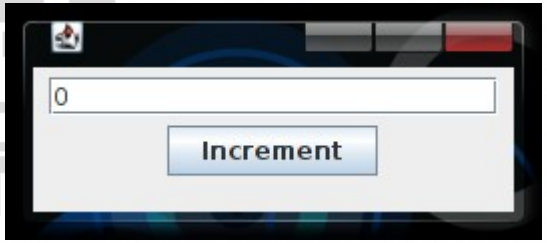


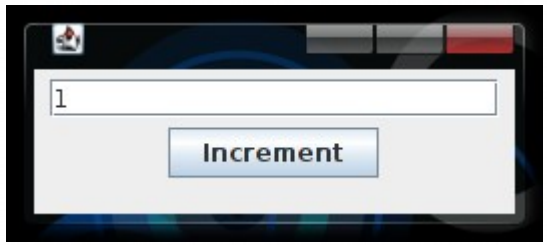
MVC – The code

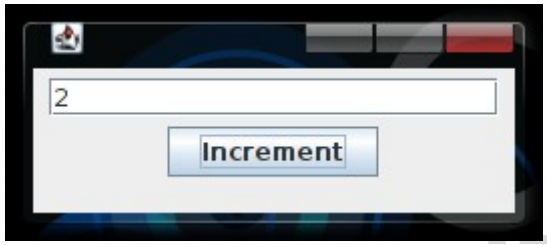
- MVC strives for Separation of Concerns and reusability of components
- As there is only one main() method in a program is is not reusable.
- As such, none of the components contain a main method()
- The main method will create the Model, Controller and View

MVC – Basic example

- Simple example: A button that increments a counter each time it's clicked







MVC Example - Model

```
public class Model {
    private int total = 0;

    public void increment() {
        total++;
    }

    public int getTotal() {
        return total;
    }
}
```

MVC Example - View

```java
public class View {
    private Model model;

    private JFrame frame;
    private JTextField text;
    private JButton button;

    public View(Controller controller, Model model) {
        this.model = model;

        controller.addView(this);

        frame = new JFrame();
        frame.setLayout(new FlowLayout());
        text = new JTextField(20);
        frame.add(text);

        button = new JButton("Increment");
        button.addActionListener(controller);

        frame.add(button);
        frame.setSize(250, 100);
        frame.setVisible(true);

        refresh();
    }

    public void refresh() {
        text.setText(Integer.toString(model.getTotal()));
    }
}
```

The view needs access to both the controller and the model

However, the controller can manage more than one view at a time

The view must be assigned to the Controller

And the view must Know about the controller

MVC Example - Controller

```
public class Controller implements ActionListener {

    private ArrayList<View> views;
    private Model model;

    public Controller(Model model) {
        this.model = model;
        this.views = new ArrayList<View>();
    }

    public void addView(View view) {
        this.views.add(view);
    }

    public void actionPerformed(ActionEvent e) {
        model.increment();
        for (View v: views) v.refresh();
    }
}
```

The controller is storing multiple views in an ArrayList

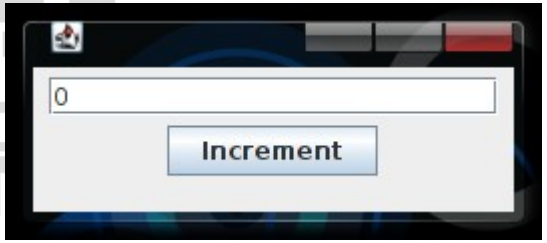
When the controller is Added to the view, It assigns itself to the controller

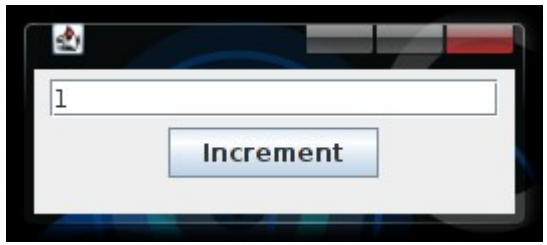
MVC – Putting it together

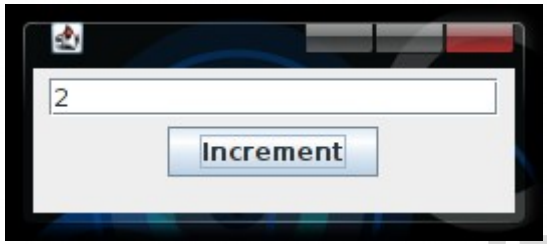
```
public class MVCExample {

    public static void main(String[] args) {
        Model model = new Model();
        Controller controller = new Controller(model);
        View view = new View(controller, model);
    }

}
```

MVC – Basic example

- For something so trivial that is a lot of code!







MVC

- However, it's very *flexible*
- Flexibility is desirable because it means making changes is easy.
- I can re-use the program, and change only the View to display the textual representation of the numbers:

MVC Example - View

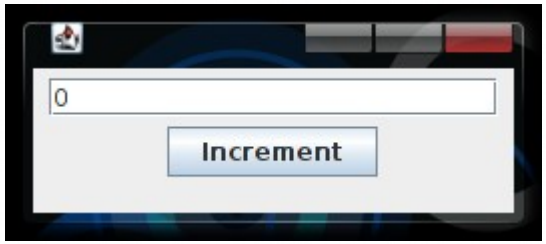
```
public class View2 {
     private Model model;

     private JFrame frame;
     private JTextField text;
     private JButton button;

     public View(Controller controller, Model model) {
          this.model = model;

          controller.addView(this);

          //…...
          refresh();
     }


     public void refresh() {
          int total = model.getTotal();
          String[] values = {"Zero", "One", "Two", "Three",
                             "Four", "Five", "Six"};
          text.setText(values[total]);
     }
}
```

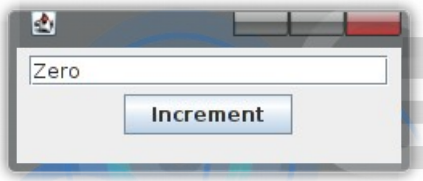
MVC Example

- I can now create a program to either display the textual or numerical representation by swapping out the view being used

```
Model model = new Model();
Controller controller = new Controller(model);
View view = new View(controller, model);
```

```
Model model = new Model();
Controller controller = new Controller(mode
View view = new View2(controller, model);
```





MVC Example

- This becomes very useful when you develop larger programs
- Larger programs tend to share pieces of functionality with other programs
- By separating the components out it allows you to easily reuse existing code in different programs
- Alternatively, you can easily implement similar functionality in the same program
- For example, a user option to decide which version of the view to display

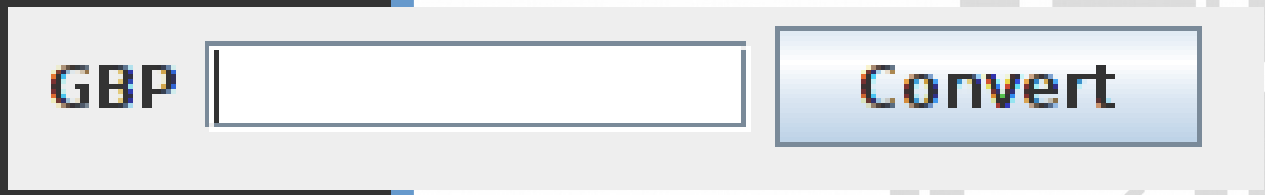
MVC – Better example

- A good use of MVC is having a program with multiple views that use the same mode/Controller
- Consider a Currency Converter that converts from £ to various other currencies
- For this component the GUI will have:
  - A label for each currency
  - A text field for each currency to show the number
  - A button to convert to other currencies

```java
public class CurrencyConverterView  {

    private JPanel panel;
    private JLabel label;
    private JTextField text;
    private JButton button;

    private String currency;

    public CurrencyConverterView(String currency) {

        this.currency = currency;

        this.panel = new JPanel();
        this.button = new JButton("Convert");

        this.label = new JLabel(currency);
        this.panel.add(label);

        this.text = new JTextField(10);

        this.panel.add(text);
        this.panel.add(button);
    }

    public JPanel getPanel() {
        return this.panel;
    }


}
```

```java
CurrencyConverterView gbpView =
    new CurrencyConverterView("GBP");
```

This creates a JPanel Instance. When the Panel is added to a Window it will look like this

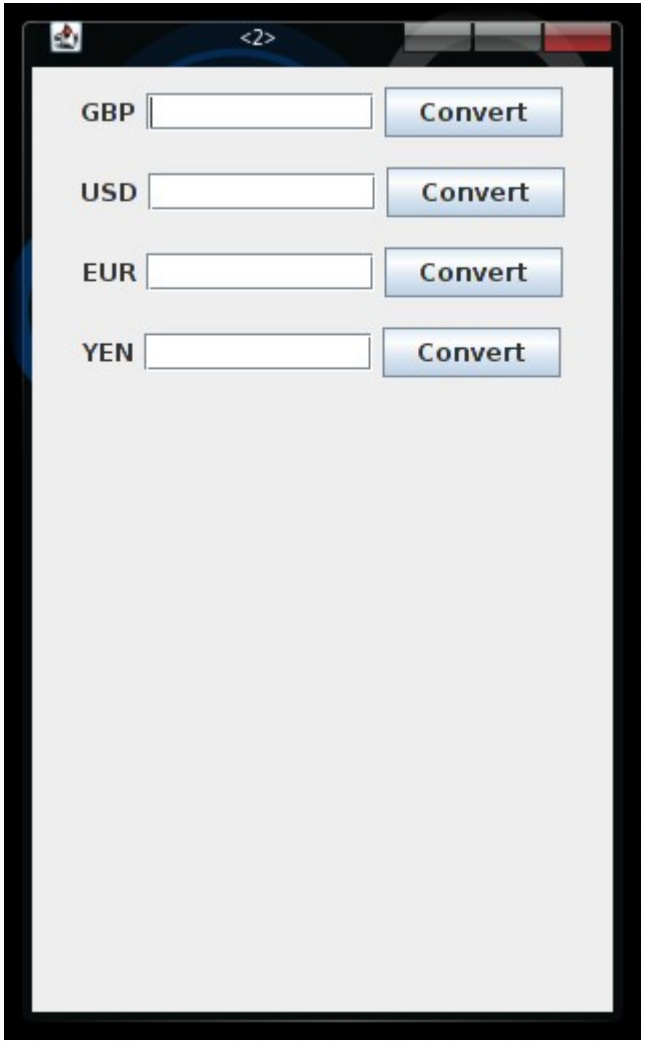


Currency Converter

- This could be reused with other currency types to build a more complete GUI:

```
public class CurrencyConverterRun {
    public static void main(String[] args) {
        JFrame window = new JFrame();
        window.setLayout(new FlowLayout());

        CurrencyConverterView gbpView = new CurrencyConverterView("GBP");
        window.add(gbpView.getPanel());

        CurrencyConverterView usdView = new CurrencyConverterView("USD");
        window.add(usdView.getPanel());

        CurrencyConverterView eurView = new CurrencyConverterView("EUR");
        window.add(eurView.getPanel());

        CurrencyConverterView yenView = new CurrencyConverterView("YEN");
        window.add(yenView.getPanel());

        window.setSize(300, 500);
        window.setVisible(true);
    }
}
```

Currency Converter



Currency Converter

- By making the Panel a reusable view, it is possible to very quickly extend the currencies supported by the application
- Clicking the buttons doesn't do anything yet
- However, all the buttons will do a similar job.
- Clicking the “Convert” button on “GBP” will take the value in the GBP text box, and fill all the other text boxes with the converted value

Currency Converter - Model

- Before any of the actions can be added, some code is needed to do the actual currency conversion

```java
public class CurrencyConverterModel {
    private double gbpValue = 0.0;
    private HashMap<String,Double> rates;

    public CurrencyConverterModel() {
        rates = new HashMap<String,Double>();
        //Exchange rates relative to GBP
        rates.put("GBP", 1.0);
        rates.put("USD", 0.6);
        rates.put("EUR", 0.83);
        rates.put("YEN", 0.0058);
    }

    public double getTotal(String currency) {
        double rate = 1/rates.get(currency);
        return this.gbpValue * rate;
    }

    public void set(String baseCurrency, double amount) {
        double rate = rates.get(baseCurrency);
        this.gbpValue = amount * rate;
    }

}
```

Currency converter

- Since the model doesn't need the GUI to run, it can be tested independently to check it works

```
CurrencyConverterModel model = new CurrencyConverterModel();

model.set("GBP", 100);
System.out.println("100 GBP in USD is " + model.getTotal("USD"));

model.set("USD", 100);
System.out.println("100 USD in GBP is " + model.getTotal("GBP"));
```

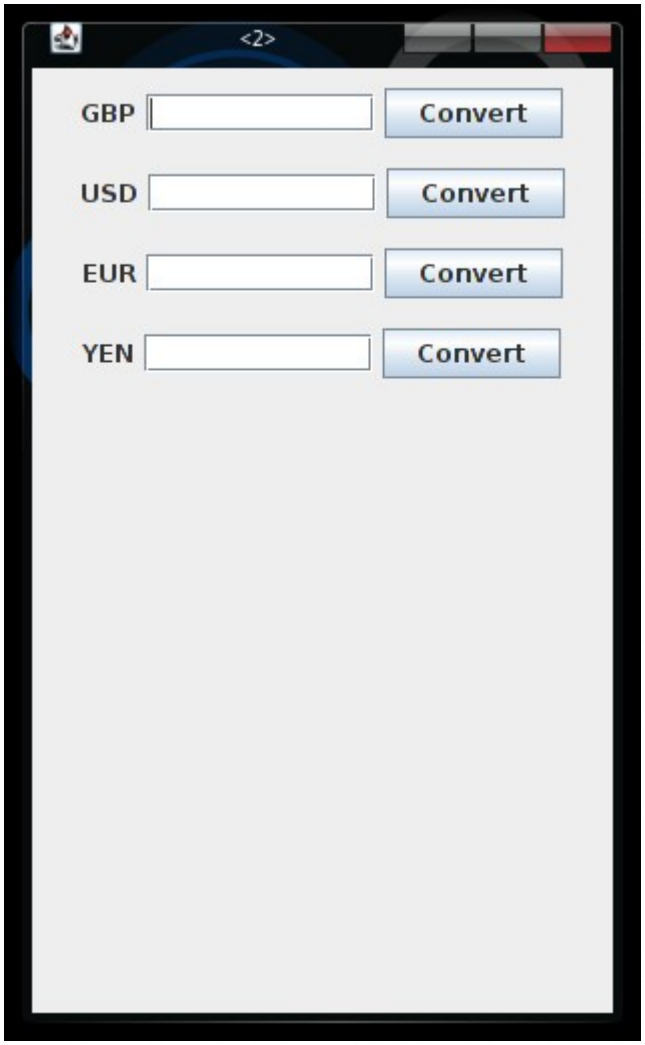
```
Output:

100 GBP in USD is 166.66666666666669
100 USD in GBP is 60.0
```

MVC - Model

- The model can be used without a GUI
- Or any GUI
- This makes the model reusable in any application which needs to convert currencies as the code for doing the conversion is completely separated from the GUI code
- This allows models to be shared between different applications or used in different ways within the same application

MVC Model

- The model can now be built into the existing GUI



The controller

- When any of the convert buttons is pressed two things need to happen:
  - 1) The model needs to be updated using the model.set method with the chosen currency and the amount being converted
  - 2) All 4 views need to be refreshed with data from the model using the model's getTotal() method

The controller

- The simplest part is refreshing the views. To do this, a refresh() method needs to be added to the view to make it read data from the model
-

```java
public class CurrencyConverterView  {

     private JPanel panel;
     private JLabel label;
     private JTextField text;
     private JButton button;

     private String currency;

     public CurrencyConverterView(String currency, CurrencyConverterModel model) {
          this.model = model;

          this.currency = currency;

          this.panel = new JPanel();
          this.button = new JButton("Convert");

          this.label = new JLabel(currency);
          this.panel.add(label);
          this.text = new JTextField(10);
          this.panel.add(text);
          this.panel.add(button);
          refresh();
     }

     public void refresh() {
          double total = this.model.getTotal(this.currency);
          this.text.setText(Double.toString(total));
     }

     public JPanel getPanel() {
          return this.panel;
     }
}
```

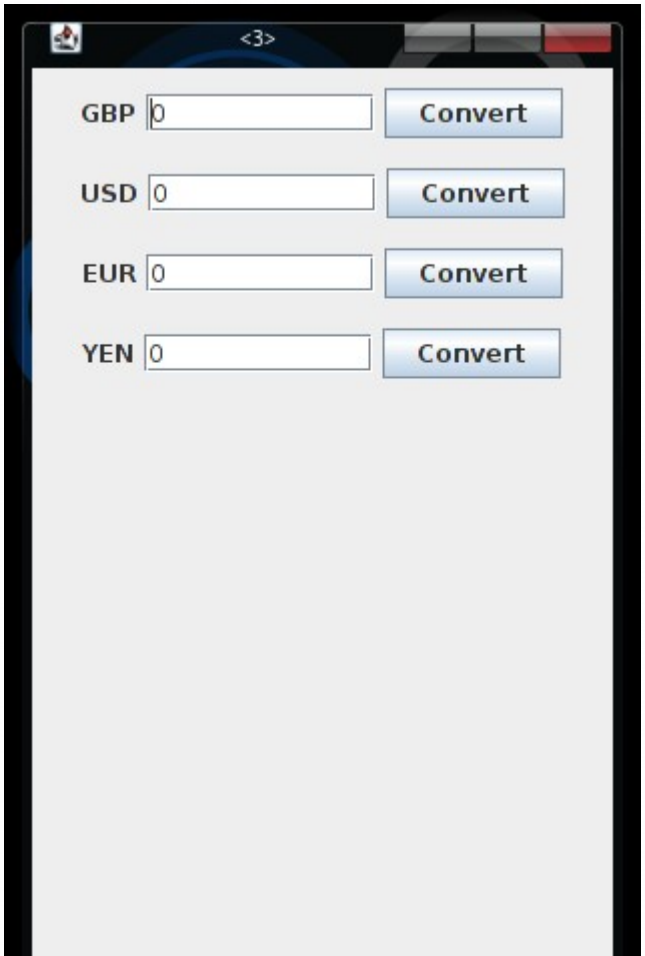
Currency Converter

- The main method needs to be updated to create the model and pass it to each of the view instances

```
JFrame window = new JFrame();
window.setLayout(new FlowLayout());

CurrencyConverterModel model = new CurrencyConverterModel();

CurrencyConverterView gbpView = new CurrencyConverterView("GBP", model);
window.add(gbpView.getPanel());

CurrencyConverterView usdView = new CurrencyConverterView("USD", model);
window.add(usdView.getPanel());

CurrencyConverterView eurView = new CurrencyConverterView("EUR", model);
window.add(eurView.getPanel());

CurrencyConverterView yenView = new CurrencyConverterView("YEN", model);
window.add(yenView.getPanel());

window.setSize(300, 500);
window.setVisible(true);
```

Currency Converter

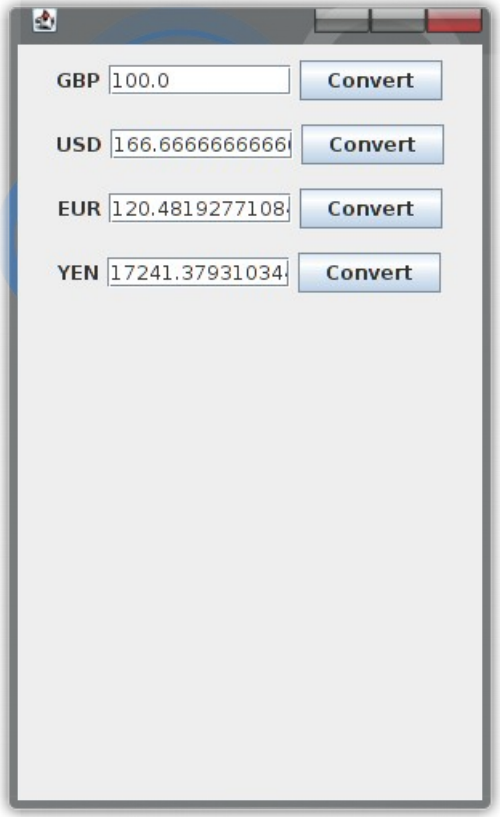
- Now that the view has access to the model and is reading the total it will load with the default total of 0:



Currency Converter

- To test it's working, a default value can be set on the model:

```
JFrame window = new JFrame();
window.setLayout(new FlowLayout());

CurrencyConverterModel model = new CurrencyConverterModel();
model.set("GBP", 100);

CurrencyConverterView gbpView = new CurrencyConverterView("GBP", model);
window.add(gbpView.getPanel());

CurrencyConverterView usdView = new CurrencyConverterView("USD", model);
window.add(usdView.getPanel());

CurrencyConverterView eurView = new CurrencyConverterView("EUR", model);
window.add(eurView.getPanel());

CurrencyConverterView yenView = new CurrencyConverterView("YEN", model);
window.add(yenView.getPanel());

window.setSize(300, 500);
window.setVisible(true);
```


GBP
100.0
Convert
USD
166.6666666666
Convert
EUR
120.4819277108
Convert
YEN
17241.37931034
Convert

Controller

- The only thing left to do is make the convert button work
- Firstly, a controller needs to be added with an action listener
- The controller needs to know about all the views so will store a HashMap of views using their currency name as the key:

```java
public class CurrencyConverterController implements ActionListener {
    private HashMap<String,CurrencyConverterView> views;
    private CurrencyConverterModel model;

    public CurrencyConverterController(CurrencyConverterModel model) {
            this.model = model;
            this.views = new HashMap<String,CurrencyConverterView>();
    }

    public void addView(String currency, CurrencyConverterView view) {
        this.views.put(currency,view);
    }
}
```

- And the views amended to ask for a controller as well as the model in their constructor:

```
public CurrencyConverterView(String currency,
                             CurrencyConverterModel model,
                             CurrencyConverterController controller) {
    this.model = model;
    this.currency = currency;
    this.controller = controller;
    this.controller.addView(currency, this);
```

- The controller is now able to refresh all the views when an action is performed

```
public class CurrencyConverterController implements ActionListener {
    private HashMap<String,CurrencyConverterView> views;
    private CurrencyConverterModel model;

    public CurrencyConverterController(CurrencyConverterModel model) {
            this.model = model;
            this.views = new HashMap<String,CurrencyConverterView>();
    }

    public void addView(String currency, CurrencyConverterView view) {
        this.views.put(currency,view);
    }

    public void actionPerformed(ActionEvent e) {
        for (CurrencyConverterView v: views.values()) v.refresh();
    }
}
```

Currency Converter

- Finally, the controller needs to be assigned as the ActionListener in the view

```java
public class CurrencyConverterView  {
     private JPanel panel;
     private JLabel label;
     private JTextField text;
     private JButton button;
     private CurrencyConverterModel model;
     private CurrencyConverterController controller;
     private String currency;

     public CurrencyConverterView(String currency,
                                  CurrencyConverterModel model,
                                  CurrencyConverterController controller) {
          this.model = model;
          this.currency = currency;
          this.controller = controller;
          this.controller.addView(currency, this);
          this.panel = new JPanel();
          this.button = new JButton("Convert");
          this.button.addActionListener(this.controller);

          this.label = new JLabel(currency);
          this.panel.add(label);
          this.text = new JTextField(10);
          this.panel.add(text);
          this.panel.add(button);

          refresh();
     }

     public void refresh() {
          double total = this.model.getTotal(this.currency);
          this.text.setText(Double.toString(total));
     }

     public JPanel getPanel() {
          return this.panel;
     }
}
```

Currency Converter

- However, the controller will need some way of knowing which "Convert" button was pressed.
  - Is it converting from pounds or euros or dollars or yen?
- Swing provides a **setActionCommand()** method on any component with a **addActionListener()** method
- The action command is a string which can be read from the ActionEvent in the actionPerformed method

```java
public class CurrencyConverterView  {
     private JPanel panel;
     private JLabel label;
     private JTextField text;
     private JButton button;
     private CurrencyConverterModel model;
     private CurrencyConverterController controller;
     private String currency;

     public CurrencyConverterView(String currency,
                                  CurrencyConverterModel model,
                                  CurrencyConverterController controller) {
          this.model = model;
          this.currency = currency;
          this.controller = controller;
          this.controller.addView(currency, this);
          this.panel = new JPanel();
          this.button = new JButton("Convert");
          this.button.addActionListener(this.controller);
          this.button.setActionCommand(this.currency);

          this.label = new JLabel(currency);
          this.panel.add(label);
          this.text = new JTextField(10);
          this.panel.add(text);
          this.panel.add(button);

          refresh();
     }

     public void refresh() {
          double total = this.model.getTotal(this.currency);
          this.text.setText(Double.toString(total));
     }

     public JPanel getPanel() {
          return this.panel;
     }
}
```

Currency Converter

- There must be a way for the controller to read data from the view. This can be provided by a method:

```java
public class CurrencyConverterView  {
     private JPanel panel;
     private JLabel label;
     private JTextField text;
     private JButton button;
     private CurrencyConverterModel model;
     private CurrencyConverterController controller;
     private String currency;

     public CurrencyConverterView(String currency,
                                  CurrencyConverterModel model,
                                  CurrencyConverterController controller) {
          this.model = model;
          this.currency = currency;
          this.controller = controller;
          this.controller.addView(currency, this);
          //….

          refresh();
     }

     public String getValue() {
          return this.text.getText();
     }

     public void refresh() {
          double total = this.model.getTotal(this.currency);
          this.text.setText(Double.toString(total));
     }

     public JPanel getPanel() {
          return this.panel;
     }
}
```

Currency Converter

- Finally, the actionPerformed method in the controller can read the ActionCommand and the value from the view and update the model accordingly

```java
public class CurrencyConverterController implements ActionListener {
    private HashMap<String,CurrencyConverterView> views;
    private CurrencyConverterModel model;

    public CurrencyConverterController(CurrencyConverterModel model) {
            this.model = model;
            this.views = new HashMap<String,CurrencyConverterView>();
    }

    public void addView(String currency, CurrencyConverterView view) {
        this.views.put(currency,view);
    }

    public void actionPerformed(ActionEvent e) {

        System.out.println(e.getActionCommand());
        CurrencyConverterView callingView = this.views.get(e.getActionCommand());

        this.model.set(e.getActionCommand(), Double.parseDouble(callingView.getValue()));

        for (CurrencyConverterView v: views.values()) v.refresh();
    }
}
```