The Model-View Approach in Java

OK, you’ve got “Hello World” running...

What now?

Assignment:

Write a program that displays the volume and the surface area for a sphere with the radius entered by the user.

A question:

How do I enter data in a Java program?

OK, let's give it a try:

```
class Sphere
{
  public static void main(String[] args)
  {
    BufferedReader console = new BufferedReader(
        new StreamReader(System.in));
    System.out.print("Enter the radius: ");
    double radius = Double.parseDouble(
           console.readLine());

    System.out.println("Radius = " + radius);
    double volume = 4.0 / 3.0 * Math.PI *
                              radius * radius * radius;
    System.out.println("Volume = " + volume);

    double area = 4.0 * Math.PI * radius * radius;
    System.out.println("Surface area = " + area);
  }
}
```

```
Enter the radius: 100
Radius = 100.0
Volume = 4188790.2047863905
Surface area = 125663.70614359173
Press any key to continue...
```

What's wrong with this program?

- This is **<u>bad design</u>**: user interface is interspersed with calculations. In any programming language, **user interface should be always separate from calculations or processing**
- Minor point: the output is ugly (too many digits) ☹

Second try:

```
import java.text.DecimalFormat;

class Sphere
{
  private static double volume(double r)
  {
    return 4.0 / 3.0 * Math.PI * r * r * r;
  }


  private static double surfaceArea(double r)
  {
    return 4.0 * Math.PI * r * r;
  }
```

Continued

Sphere (cont'd):

```
  public static void main(String[] args)
  {
    EasyReader console = new BufferedReader(
        new StreamReader(System.in));
    System.out.print("Enter the radius: ");
    double radius = Double.parseDouble(
        console.readLine());
    DecimalFormat f3 = new DecimalFormat("0.000");

    System.out.println();

    System.out.println("Radius = " +
                                  f3.format(radius));
    System.out.println("Volume = " +
                          f3.format(volume(radius)));
    System.out.println("Surface area = " +
                      f3.format(surfaceArea(radius)));
    System.out.println();
  }
}
```

```
Enter the radius: 100

Radius = 100.000
Volume = 4188790.205
Surface area = 125663.706

Press any key to continue...
```

So far, so good...

- The output looks better
- Passable for **procedural** programming style

... But...

... this is not OOP:

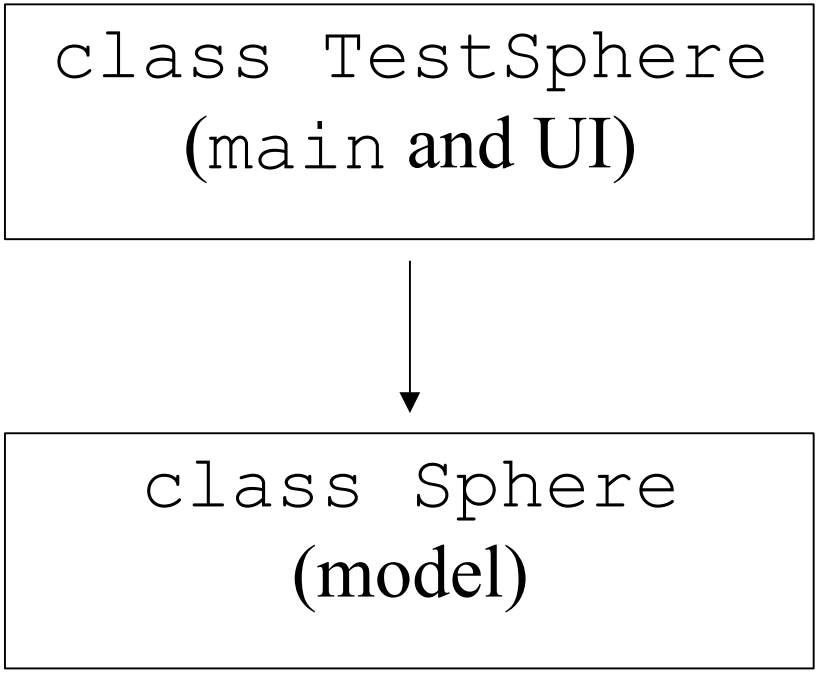
- The calculations are bunched together with the user interface (UI) <u>in the same class</u>

- It will be hard to reuse the same formulas ("methods") with a different UI

In OOP...

- Each object must have its own responsibilities: one is a model of a sphere, another implements UI
- We should be able to work as a team, each of us working on different classes

Solution: put the "model" and the UI into separate classes.



```
class Sphere
{
  private double myRadius;
  private double myCenterX;
  private double myCenterY;

  // Constructors:
  public Sphere (double x, double y, double r)
  {
    myCenterX = x;
    myCenterY = y;
    myRadius = r;
  }

  // ... other constructors
```

private fields
(data members)

Continued

Sphere (cont'd)

```
// Accessors:
public double getRadius()
{
  return myRadius;
}

// ... other accessors

// Modifiers:
public void setRadius(double r)
{
  myRadius = r;
}

// ... other modifiers
```

Continued

Sphere (cont'd)


```
  public double volume()
  {
    return 4.0 / 3.0 * Math.PI * myRadius *
                                 myRadius * myRadius;
  }

  public double surfaceArea()
  {
    return 4.0 * Math.PI * myRadius * myRadius;
  }

  // ... Other public and private methods
  public String toString()
  {
    return ”Sphere [Center = (" + myCenterX + ", "
                + myCenterY + ") Radius = " + myRadius
                + "]";
  }
}
```

TestSphere

```
import java.text.DecimalFormat;

class TestSphere
{
  public static void main(String[] args)
  {
    BufferedReader console = new BufferedReader(
        new StreamReader(System.in));
    System.out.print("Enter the radius: ");
    double radius = Double.parseDouble(
       console.readLine());

    DecimalFormat f3 = new DecimalFormat("0.000");

    Sphere balloon = new Sphere(0, 0, radius);
```

Continued

TestSphere(cont'd)

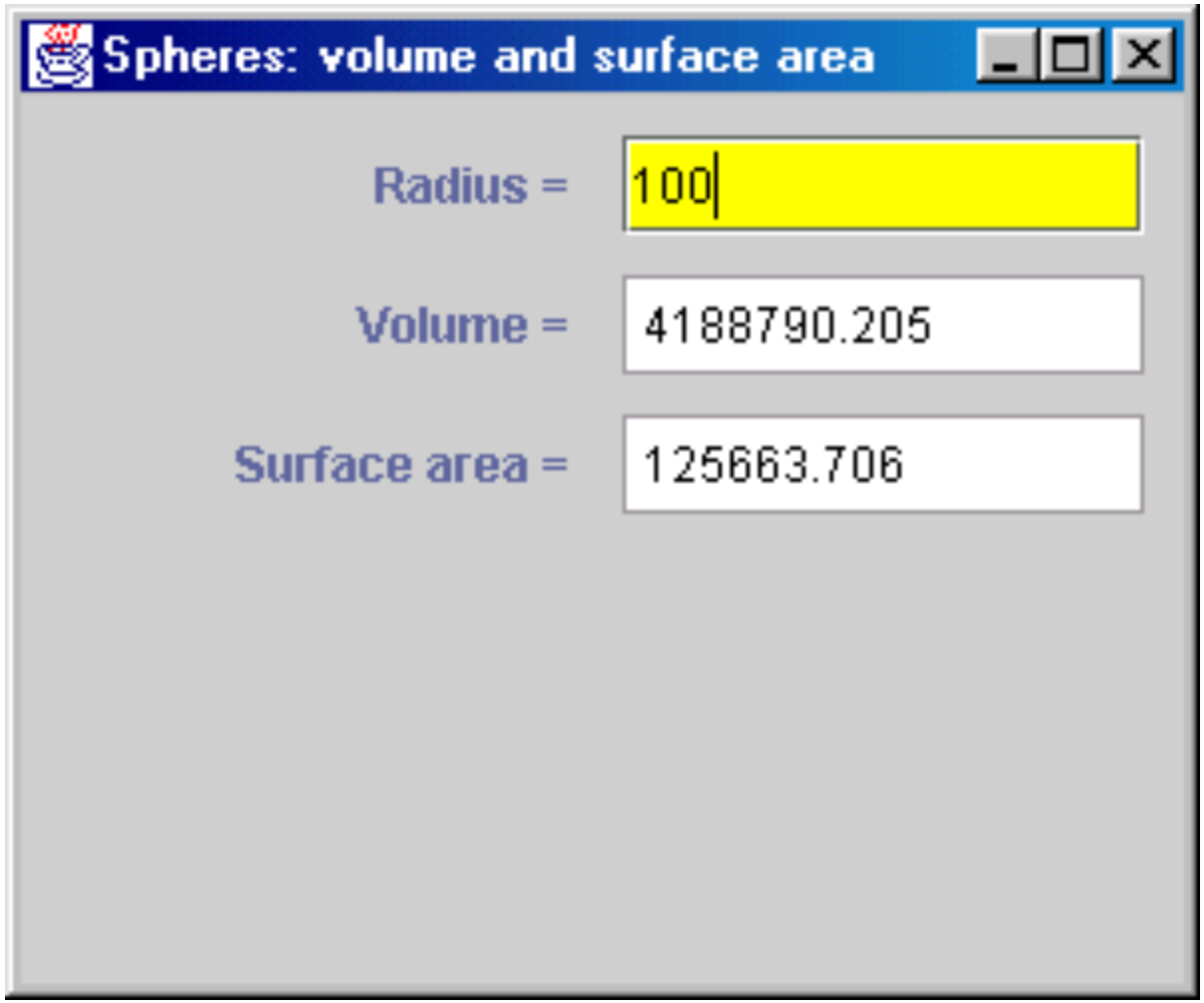
```
    System.out.println();

    System.out.println(balloon);
    System.out.println("Volume = " +
                          f3.format(balloon.volume()));
    System.out.println("Surface area = " +
                     f3.format(balloon.surfaceArea()));

    System.out.println();
  }
}
```

```
Enter the radius: 100

Sphere [Center = (0.0, 0.0) Radius = 100.0]
Volume = 4188790.205
Surface area = 125663.706

Press any key to continue...
```

Reasonable OOP design, but...

... where’s the GUI?

We want something like this:



Let's make it a team effort

- You — **a student** — write the "model" from the given specs:

```
class Sphere
{
  public Sphere (double x, double y, double r)...
  public double getRadius()...
  public void setRadius(double r)...
  public double volume()...
  public double surfaceArea()...
  public String toString()...
  etc.
}
```

Team effort...

- Another student can write the GUI

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.text.DecimalFormat;

public class SphereWindow extends JFrame
    implements ActionListener
{
  private JTextField radiusIn, volumeOut, areaOut;
  private Sphere balloon;
  private DecimalFormat f3 =
                          new DecimalFormat("0.000");
  public SphereWindow()
  {
    super("Spheres: Volume and Surface");

    JPanel view = new JPanel();
    view.setLayout(new GridLayout(3, 2, 10, 10));
    view.setBorder(new EmptyBorder(10, 10, 10, 10));

    view.add(new JLabel("Radius = ",
                    SwingConstants.RIGHT));

    radiusIn = new JTextField(8);
    radiusIn.setBackground(Color.yellow);
```

... *continued*

The GUI class `SphereWindow`

- **It is pretty straightforward but verbose**
- **It uses Java's event-handling model**
- **If you are a bright, inquisitive student, it will give you a Swing GUI example that you can use in other projects**
- **Do you <u>really</u> want to see it?**

The GUI class `SphereWindow`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.text.DecimalFormat;

public class SphereWindow extends JFrame
    implements ActionListener
{
  private JTextField radiusIn, volumeOut, areaOut;
  private Sphere balloon;
  private DecimalFormat f3 =
                 new DecimalFormat("0.000");
```

Continued ⇧

SphereWindow (cont'd)

```
public SphereWindow()
{
  super("Spheres: volume and surface area");

  JPanel view = new JPanel();
  view.setLayout(new GridLayout(6, 2, 10, 10));
  view.setBorder(new EmptyBorder(10, 10, 10, 10));

  view.add(new JLabel("Radius = ", SwingConstants.RIGHT));
  radiusIn = new JTextField(8);
  radiusIn.setBackground(Color.yellow);
  radiusIn.addActionListener(this);
  view.add(radiusIn);

  view.add(new JLabel("Volume = ", SwingConstants.RIGHT));
  ...
```

What can we learn from this?

- OOP design with a separate model (`Sphere`) and view (`SphereWindow`)
- Implementing a properly encapsulated, reusable class (`Sphere`)
- Team development
- Elements of Swing - learn by “diving into it”

Good job!

- Good OOP style
- The model and view are separate

Now, for the rest of the story...”

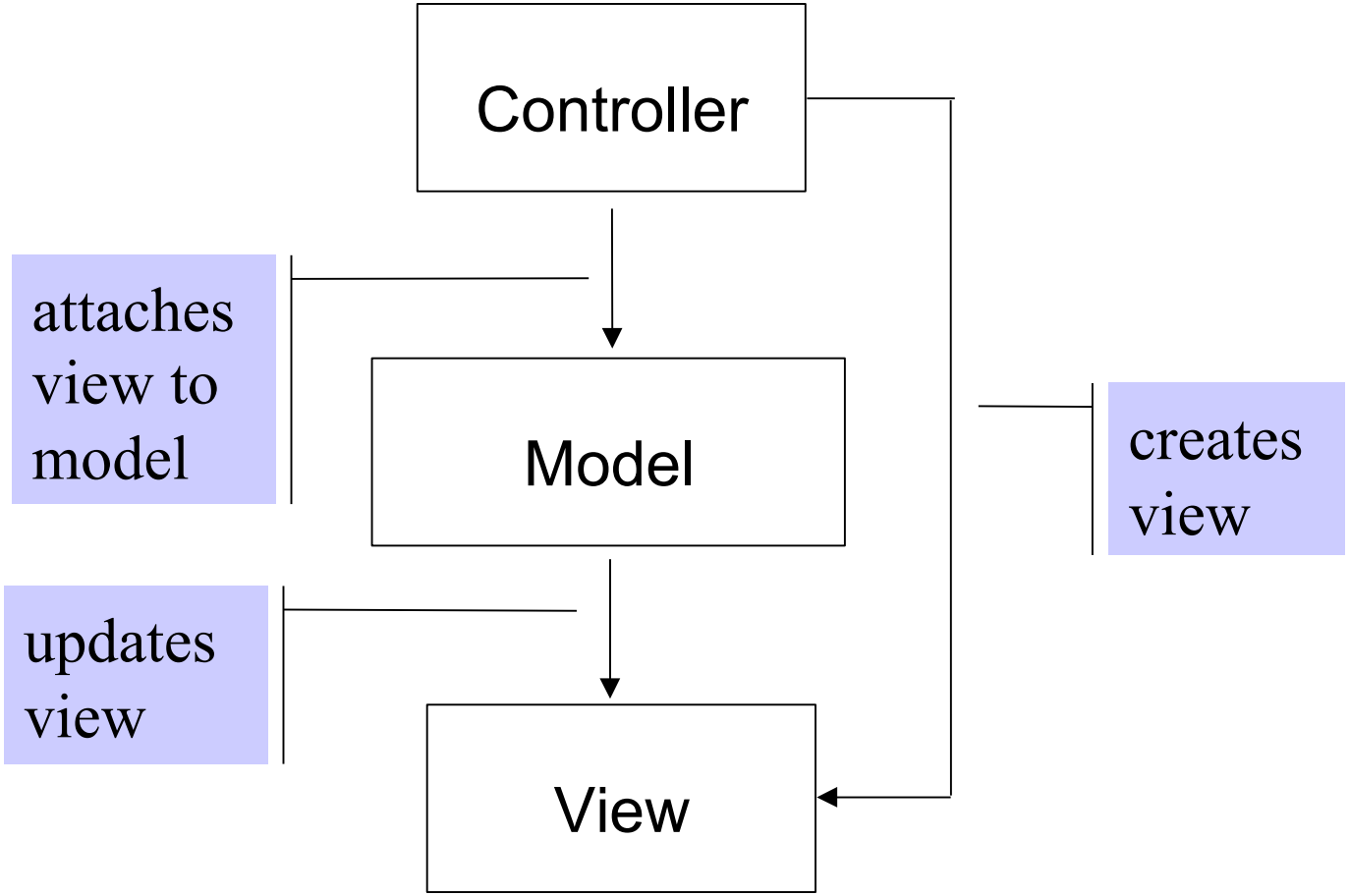
The Model-View-Controller (MVC) “design pattern”

Design Patterns

- OOP design is not easy
- Design patterns offer standard ideas for laying out classes
- MVC is a commonly used design pattern for implementing interactions between “model,” “view,” and “controller” classes

MVC — the general idea

- The controller is an object that processes user commands and program events
- The controller (or the “main” class) creates the model
- The controller creates a “view” object (or several views) and attaches it (or them) to the model
- The controller changes the state of the model
- When the model’s state changes, the model updates all the “views” attached to it


Controller
attaches view to model
Model
creates view
updates view
View

Our “Sphere” example now has three classes:

- `Sphere.java` (model) 64 lines
- `TextView.java` (view) 55 lines
- `SphereWindow.java` (controller/main) 40 lines

Hmm...

We started with only one class,
16 lines...

Now we are like real pros!
(MVC and all...)

Java supports MVC with its `Observable` library class and `Observer` interface

- A "model" class <u>extends</u> `Observable`, which provides methods for attaching observers and notifying them when a change occurs
- A "view" class <u>implements</u> `Observer` and must supply the `update` method, called automatically when the model changes

MVC implemented:

- The only changes in the `Sphere` class:

```
import java.util.Observable;

class Sphere extends Observable
{
  ...
  public void setRadius(double r)
  {
    myRadius = r;
    setChanged();
    notifyObservers();
  }
  ...
}
```

MVC implemented (cont'd)

- `SphereWindow`, the "main" class, works as controller, creates the model and the view:

```
public class SphereWindow extends JFrame
    implements ActionListener
{
  public SphereWindow()
  {
    super("Spheres: volume and surface area");

    Sphere model = new Sphere(0, 0, 100);
    TextView view = new TextView();
    model.addObserver(view);
    ...
  }
  public static void main(...
```

MVC implemented (cont'd)

- Here the main class also acts as the controller and processes GUI events:

```
public class SphereWindow extends JFrame
    implements ActionListener
{
  ...
  public void actionPerformed(ActionEvent e)
  {
    JTextField t = (JTextField)e.getSource();
    double r = Double.parseDouble(t.getText());
    model.setRadius(r);
  }
  ...
```

MVC implemented (cont'd)

- The "view" object sets up the display:

```
public class TextView extends JPanel
    implements Observer
{
  private JTextField radiusIn, volumeOut, areaOut;
  private DecimalFormat f3 =
            new DecimalFormat("0.000");

  public TextView()
  {
    setLayout(new GridLayout(6, 2, 10, 10));
    setBorder(new EmptyBorder(10, 10, 10, 10));
    add(new JLabel("Radius = ", SwingConstants.RIGHT));
    radiusIn = new JTextField(8);
    ...
```

MVC implemented (cont'd)

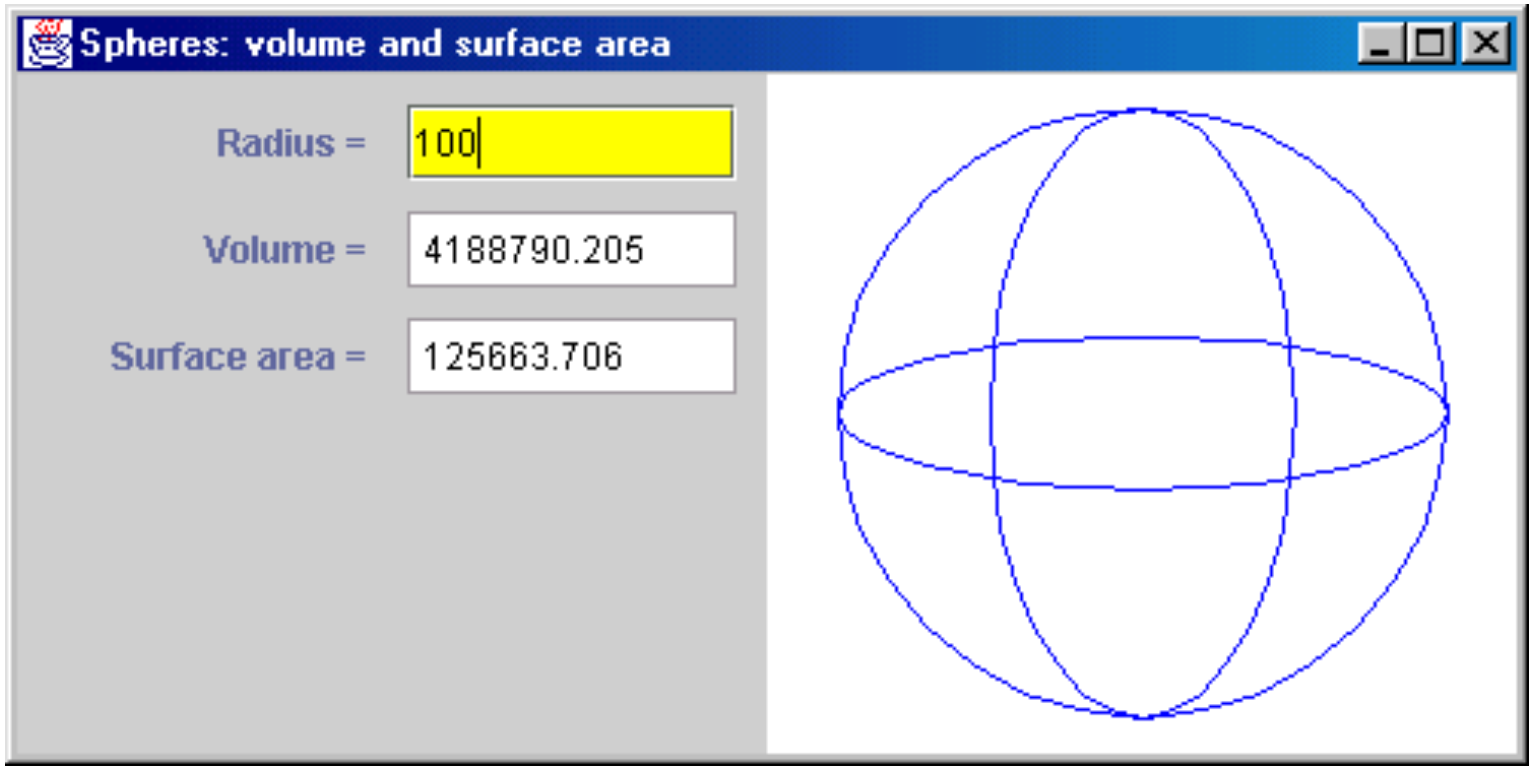
- The "view" object also updates the display when the model's state changes:

```
public class TextView extends JPanel
     implements Observer
{
  ...
  public void update(Observable o, Object arg)
  {
    Sphere balloon = (Sphere)o;
    radiusIn.setText(" " +
                     f3.format(balloon.getRadius()));
    volumeOut.setText(" " +
                     f3.format(balloon.volume()));
    ...
  }
  ...
```

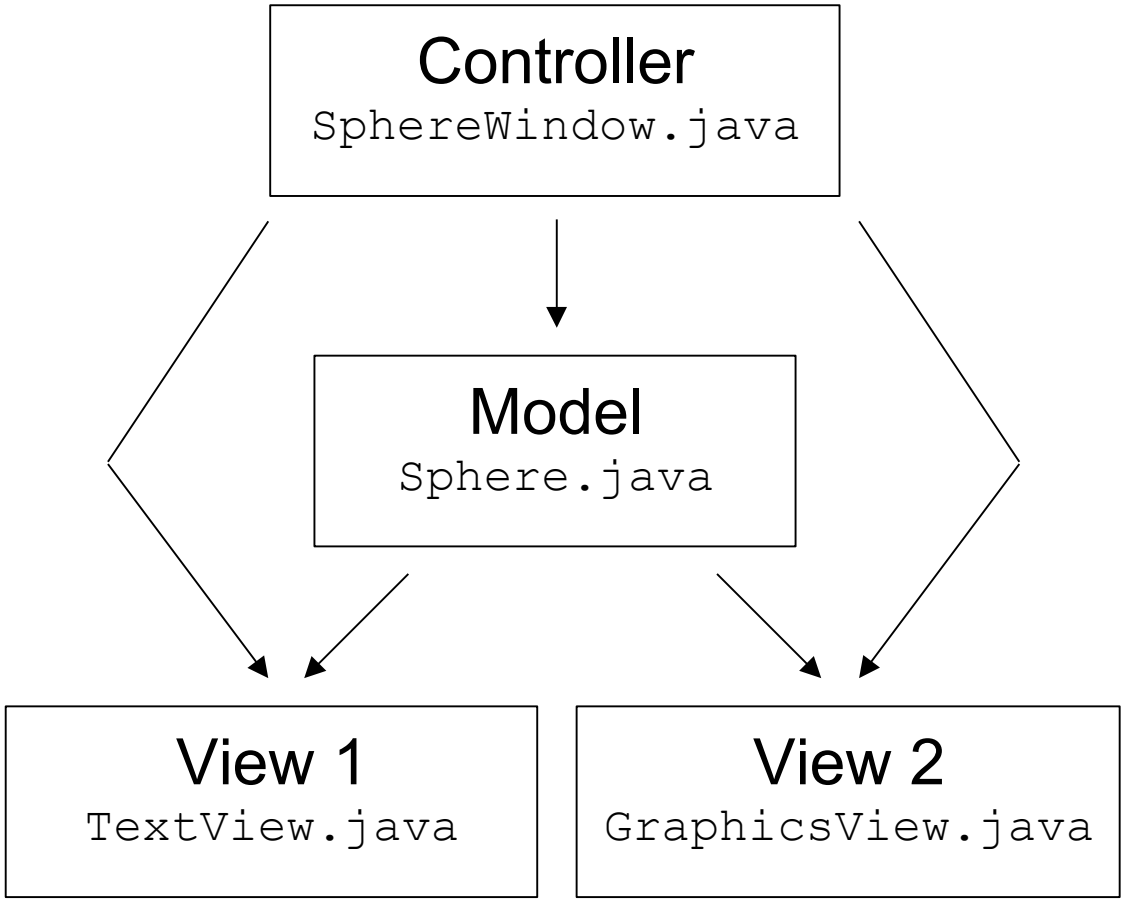
The MVC design pattern adds flexibility:

- We can easily implement several views of the same model
- We can have several controllers
- All views are updated automatically when the model changes
- All controllers work independently of each other

One model, two views



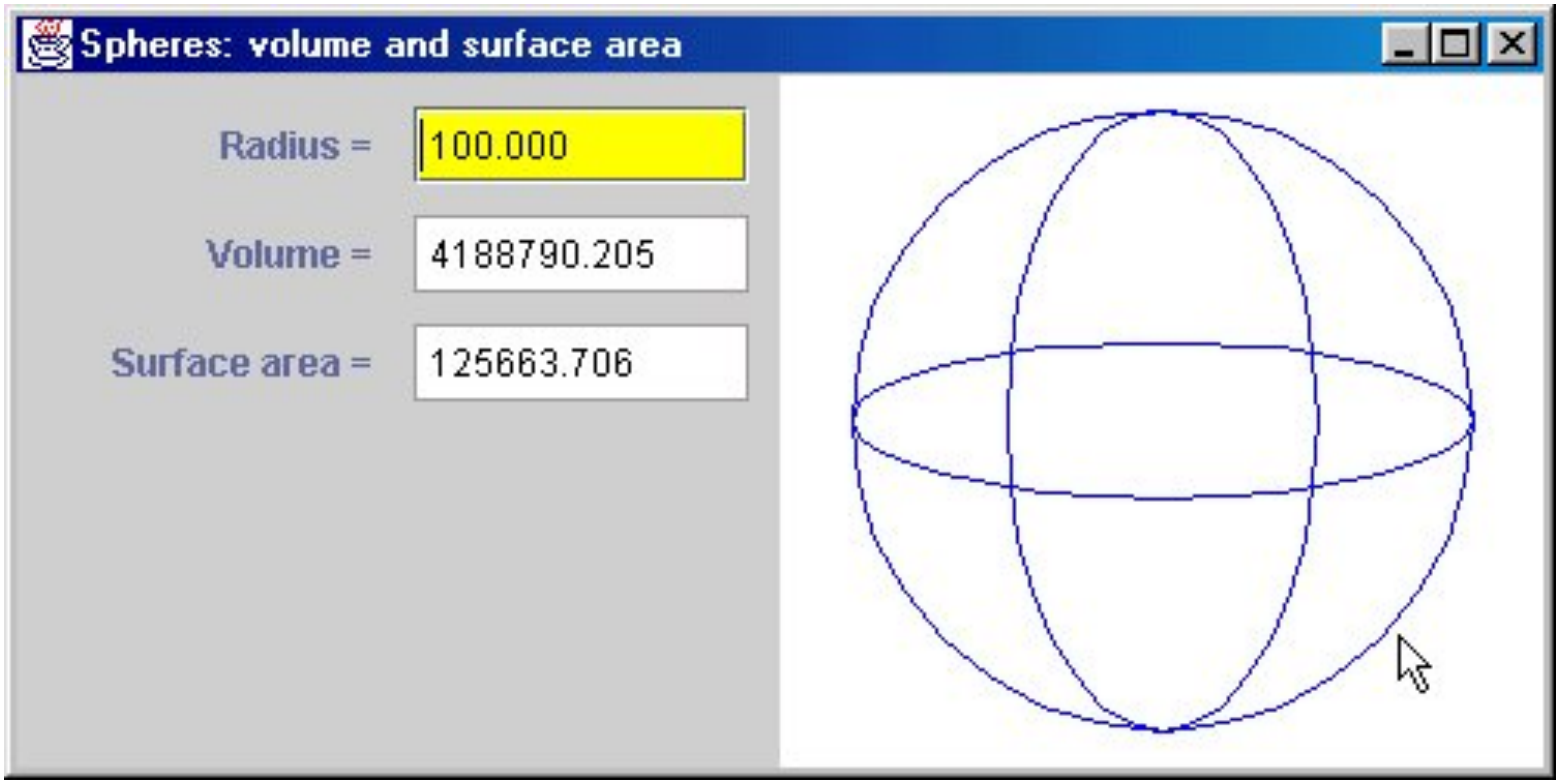
When the user enters a new radius, both the text and the graphics displays are updated.


Controller
SphereWindow.java
Model
Sphere.java
View 1
TextView.java
View 2
GraphicsView.java

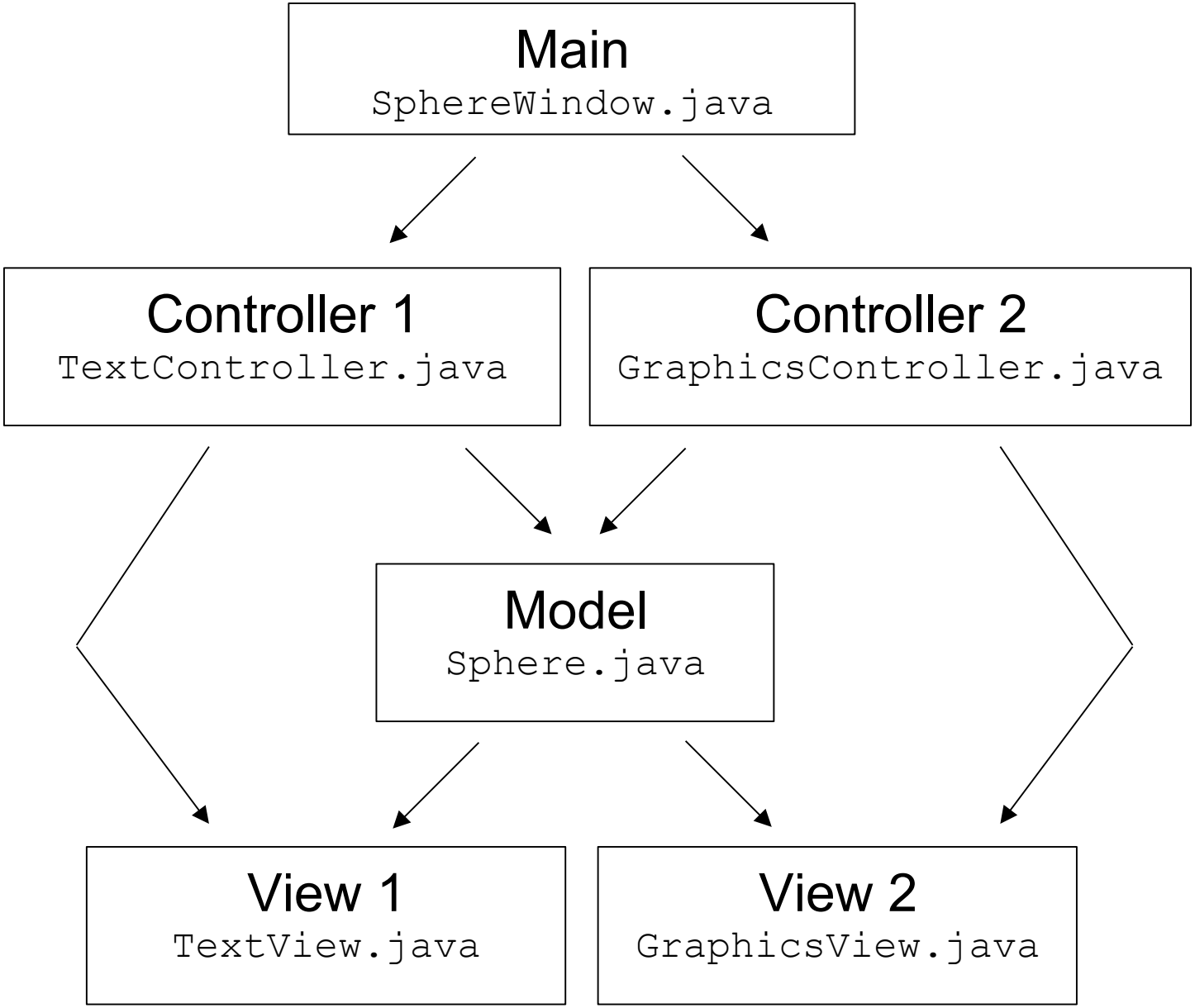
One model, two views:

```
public class SphereWindow extends JFrame
    implements ActionListener
{
  private Sphere model;

  public SphereWindow()
  {
    super("Spheres: volume and surface area");

    model = new Sphere(0, 0, 100);

    TextView tView = new TextView();
    model.addObserver(tView);
    tView.addActionListener(this);
    tView.update(model, null);

    GraphicsView gView = new GraphicsView();
    model.addObserver(gView);
    gView.update(model, null);
    ...
```

One model, two views, two controllers:



The user can either enter a new radius or stretch/squeeze the sphere with a mouse — both the text and the graphics displays are updated.


Main
SphereWindow.java
Controller 1
TextController.java
Controller 2
GraphicsController.java
Model
Sphere.java
View 1
TextView.java
View 2
GraphicsView.java