Model-view-controller

Model-View-Controller

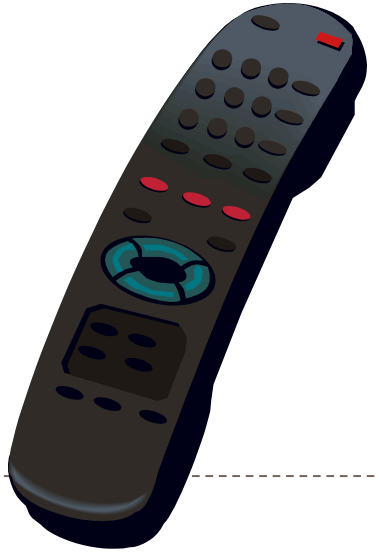
- model
  - represents state of the application and the rules that govern access to and updates of state
- view
  - presents the user with a sensory (visual, audio, haptic) representation of the model state
- controller
  - processes and responds to events (such as user actions) from the view and translates them to model method calls

Model—View—Controller

TV
- on : boolean - channel : int - volume : int
+ power(boolean) : void + channel(int) : void + volume(int) : void

Model

View

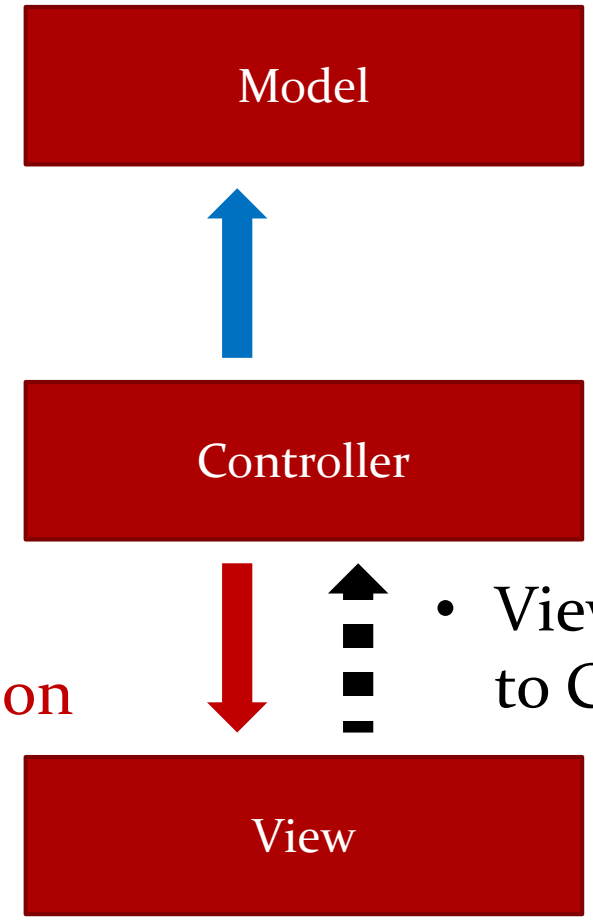


Controller

RemoteControl
+ togglePower() : void + channelUp() : void + volumeUp() : void

MVC (Notes version)

- controller invokes Model methods to:
  - get Model state
  - modify Model state
- controller invokes View methods to:
  - update the View based on changes in the Model

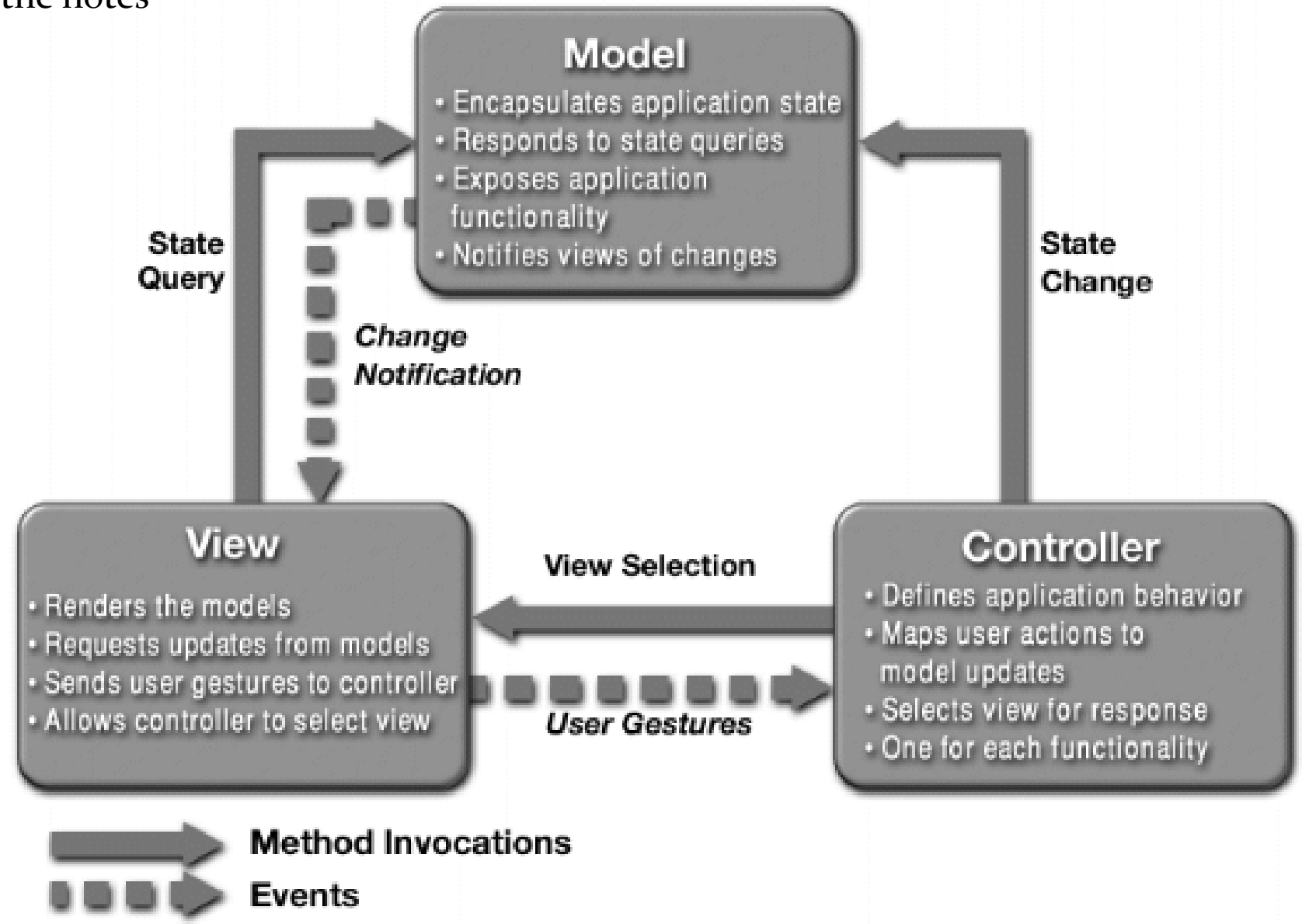


- View sends events to Controller

MVC

- why adopt this design?
- completely decouples the Model and the View

a different MVC structure
than in the notes



An Image Viewer using MVC

- allows user to load multiple images and cycle through the images using buttons

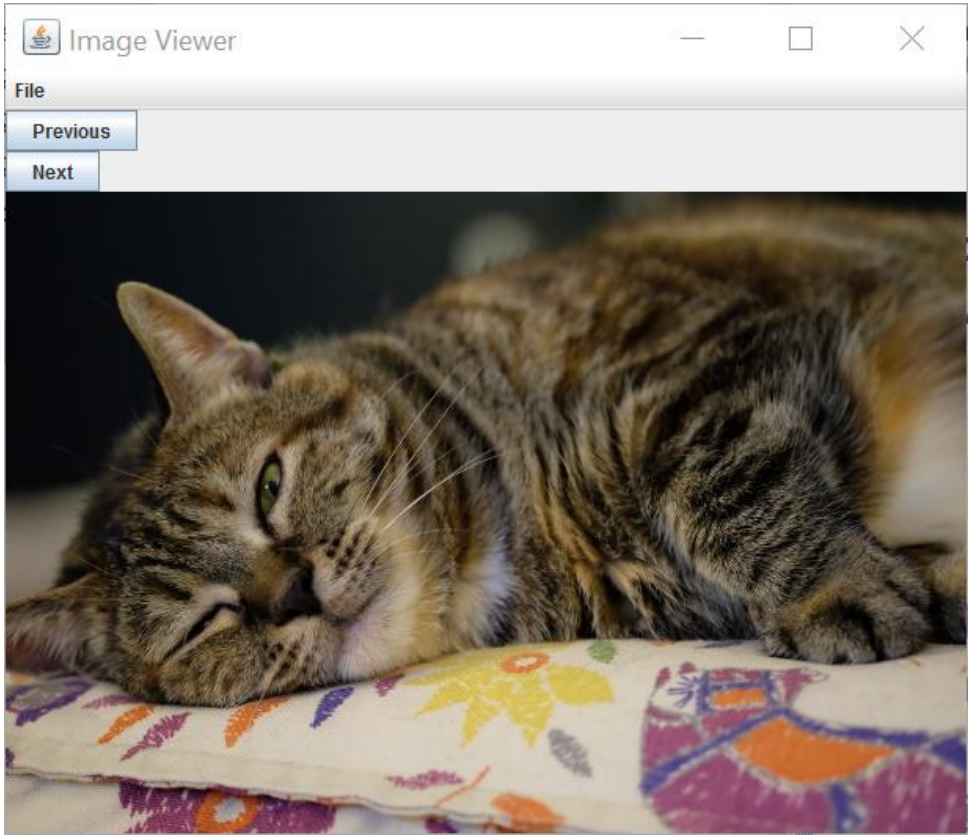


Image Viewer: Model

- model
  - the data
  - methods that get the data (accessors)
  - methods that modify the data (mutators)
- the data
  - zero or more images
- accessors
  - get the current/next/previous image
- mutators
  - load images
  - remove images

Image Viewer: Model

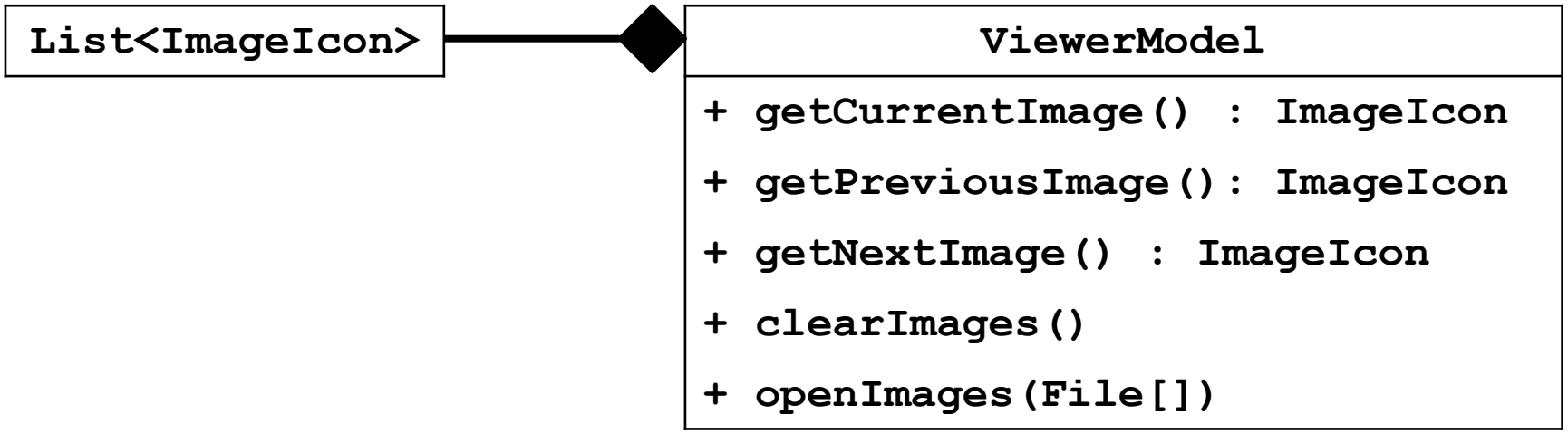


Image Viewer: Model

this.images (list of images)

0 1 2 3

this.currentImage
(index into this.images)

```
public class ViewerModel {
  private static final ImageIcon EMPTY_IMAGE = new ImageIcon();

  private int currentImage;
  private List<ImageIcon> images;

  /**
   * Creates a model having zero images.
   */
  public ViewerModel() {
    this.images = new ArrayList<ImageIcon>();
    this.currentImage = -1;
  }
```

index of the current image; intially set to -1 because there is no current image

```
/**
 * Get the current image in the model. Returns an empty image if the model
 * has no images.
 *
 * @return the current image if the model has one or more images, or an
 *         empty image otherwise
 */
public ImageIcon getCurrentImage() {
```

worksheet problem 2(a)

```
}
```

```
/**
 * Get the next image in the model and makes the next image the current
 * image. Returns the current image if the current image is the last image
 * in the model. Returns an empty image if the model has no images.
 *
 * @return the next image in the model
 */
public ImageIcon getNextImage() {
```

worksheet problem 2(b)

```
}
```

```java
/**
 * Get the previous image in the model and makes the previous image the
 * current image. Returns the current image if the current image is the
 * first image in the model. Returns an empty image if the model has no
 * images.
 *
 * @return the previous image in the model
 */
public ImageIcon getPreviousImage() {
  ImageIcon i = EMPTY_IMAGE;
  if (this.currentImage > 0) {
    this.currentImage--;
  }
  if (this.currentImage >= 0) {
    i = this.images.get(this.currentImage);
  }
  return i;
}
```

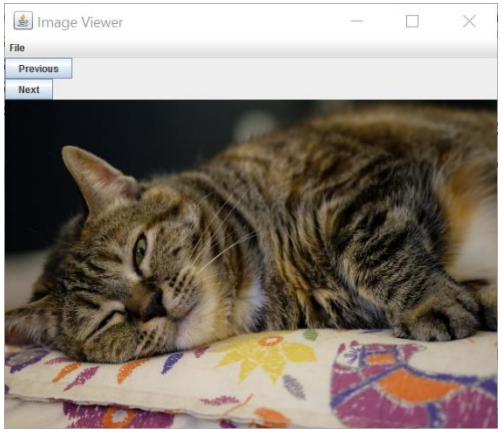
```
/**
 * Clear the images held by the model.
 */
public void clearImages() {
  this.currentImage = -1;
  this.images.clear();
}
```

```java
/**
 * Read an array of image files into the model. The images are added to the
 * end of the sequence of images in the model. Image file formats supported
 * include JPEG, PNG, and GIF.
 *
 * @param files
 *            an array of <code>File</code> references to open
 */
public void openImages(File[] files) {
  List<ImageIcon> icons = new ArrayList<ImageIcon>();
  for (File f : files) {
    ImageIcon icon = new ImageIcon(f.getAbsolutePath());
    if (icon.getImageLoadStatus() == MediaTracker.COMPLETE) {
      icons.add(icon);
    }
  }
  if (this.currentImage < 0 && icons.size() > 0) {
    this.currentImage = 0;
  }
  this.images.addAll(icons);
}
```

Image Viewer: View

- view
  - a visual (or other) display of the model
  - a user interface that allows a user to interact with the view
  - methods that get information from the view (accessors)
  - methods that modify the view (mutators)
- a visual (or other) display of the model
  - the current image
- a user interface that allows a user to interact with the view
  - next and previous buttons
  - a menu to load and clear images

Image Viewer: View



```
public class ViewerView extends JFrame {
```

does not implement ActionListener (although it could) because the controller will listen for events and respond accordingly

```
public class ViewerView extends JFrame {

   private JLabel img;
   private JButton next;
   private JButton prev;
   private JFileChooser fc;
```

label displays the image
button for next image
button for previous image
a file chooser to allow the user to choose the images to show

```
/**
 * String for the action command for opening files.
 */
public static final String OPEN = "OPEN";

/**
 * String for the action command for clearing all of the images.
 */
public static final String CLEAR = "CLEAR";

/**
 * String for the action command for showing the previous image.
 */
public static final String PREVIOUS = "PREVIOUS";

/**
 * String for the action command for showing the next image.
 */
public static final String NEXT = "NEXT";
```

```
public ViewerView(ActionListener listener) {
  // 1. Create the frame
  super("Image Viewer");

  // 2. Choose what happens when the frame closes
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  // 3. Create components and put them in the frame
  this.makeLabel();
  this.makeFileChooser();
  this.makeMenu(listener);
  this.makeButtons(listener);
  this.add(this.prev);
  this.add(this.next);
  this.add(this.img);

  // 4. Size the frame
  this.setMinimumSize(new Dimension(600, 400));
  this.pack();
  this.setLayout(new BoxLayout(this.getContentPane(), BoxLayout.Y_AXIS));

  // 5. Show the view
  // we'll let ViewerApp do this
}
```

the controller will be the listener for events

```
private void makeMenu(ActionListener listener) {
   JMenuBar menuBar = new JMenuBar();

   JMenu fileMenu = new JMenu("File");
   menuBar.add(fileMenu);

   JMenuItem openMenuItem = new JMenuItem("Open...");
   openMenuItem.setActionCommand(ViewerView.OPEN);
   openMenuItem.addActionListener(listener);
   fileMenu.add(openMenuItem);

   JMenuItem clear = new JMenuItem("Clear");
   clear.setActionCommand(ViewerView.CLEAR);
   clear.addActionListener(listener);
   fileMenu.add(clear);

   this.setJMenuBar(menuBar);
}
```

the controller will be the listener

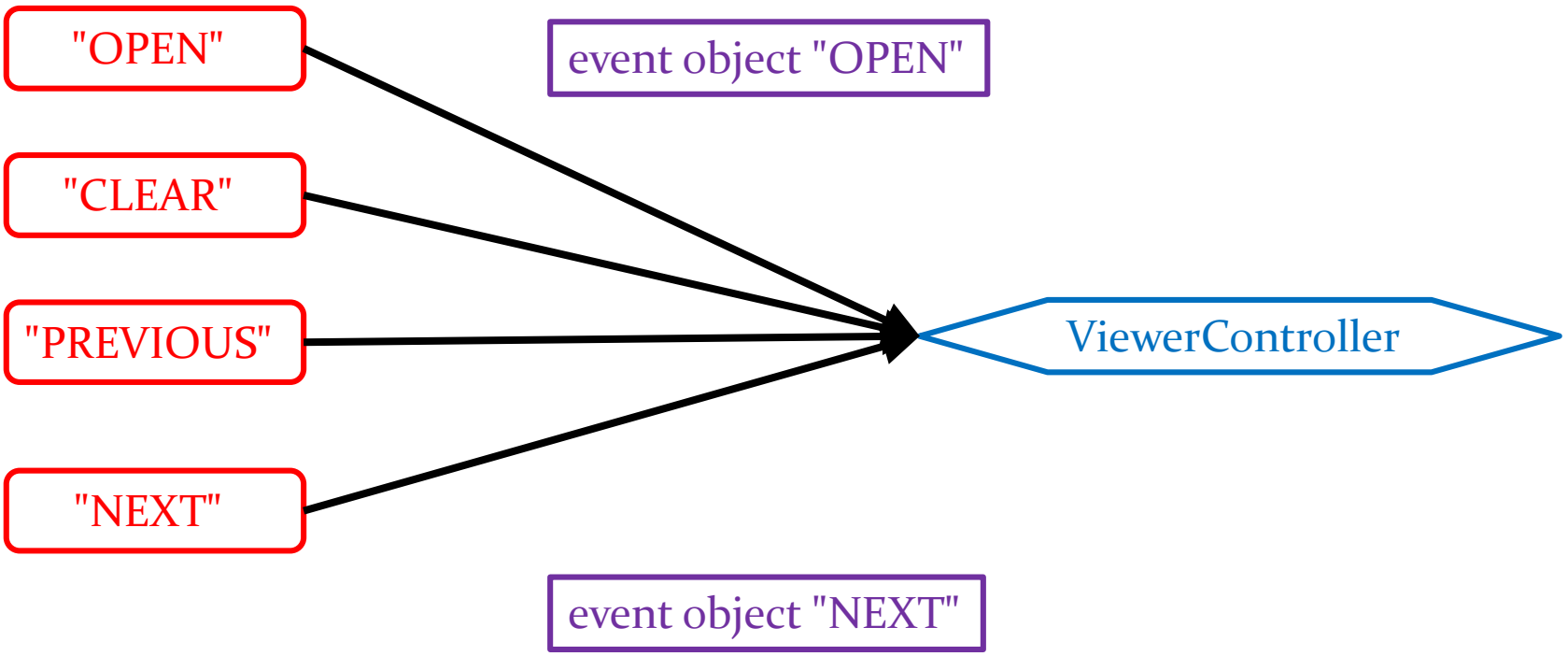
```
private void makeButtons(ActionListener listener) {
   this.prev = new JButton("Previous");
   this.prev.setActionCommand(ViewerView.PREVIOUS);
   this.prev.addActionListener(listener);

   this.next = new JButton("Next");
   this.next.setActionCommand(ViewerView.NEXT);
   this.next.addActionListener(listener);
}
```

the controller will be the listener

```
/**
 * Sets the image shown by the view to the specified image.
 *
 * @param icon
 *             an image to show in the view
 */
public void setImage(ImageIcon icon) {
  if (this.img.getIcon() != icon) {
    this.img.setIcon(icon);
    this.pack();
  }
}
```

change the image only if the image to show is a different ImageIcon object than what is currently in the label

Not a UML Diagram



Controller

- controller
  - processes and responds to events (such as user actions) from the view and translates them to model method calls
- needs to interact with both the view and the model but does not own the view or model
  - aggregation

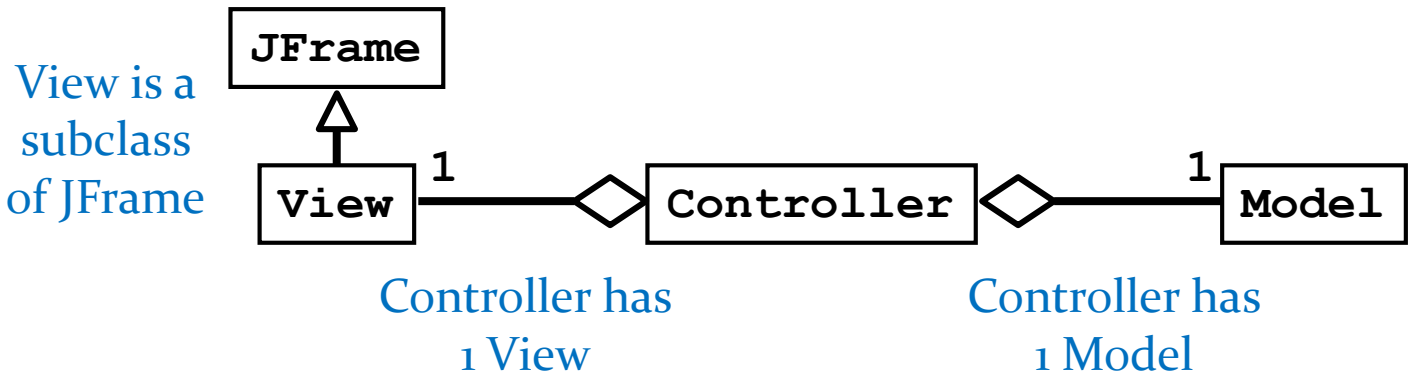


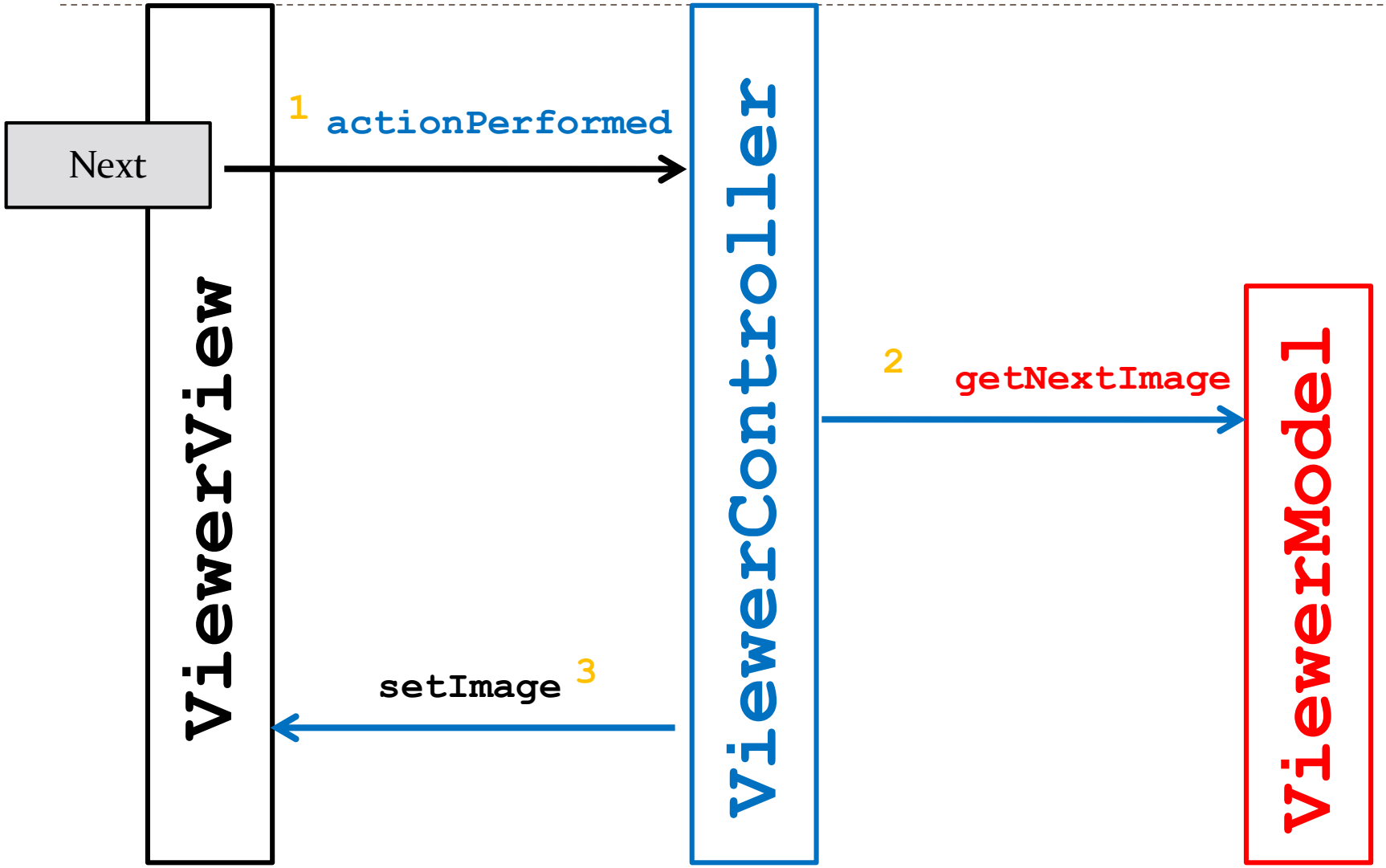
Image Viewer Controller

- recall that our application only uses events that are fired by buttons (**`Jbuttons`** and **`JMenuItems`**)
  - a button fires an **`ActionEvent`** event whenever it is clicked
- **`ViewerController`** listens for fired **`ActionEvents`**
  - how? by implementing the **`ActionListener`** interface

```
public interface ActionListener
{
  void actionPerformed(ActionEvent e);
}
```

- **`ViewerController`** was registered to listen for **`ActionEvent`**s fired by the various buttons in **`ViwerView`**
- whenever a button fires an event, it passes an **`ActionEvent`** object to **`ViewerController`** via the **`actionPerformed`** method
  - **`actionPerformed`** is responsible for dealing with the different actions (open, save, sum, etc)

User Presses Next Button



```
public class ViewerController implements ActionListener {

  private ViewerModel model;
  private ViewerView view;

  /**
   * Creates a controller having no model and no view.
   */
  public ViewerController() {
  }
```

```
/**
 * Sets the model for the controller.
 *
 * @param model
 *             the viewer model the controller should use
 */
public void setModel(ViewerModel model) {
  this.model = model;
}

/**
 * Sets the view for the controller.
 *
 * @param view
 *             the view the controller should use
 */
public void setView(ViewerView view) {
  this.view = view;
}
```

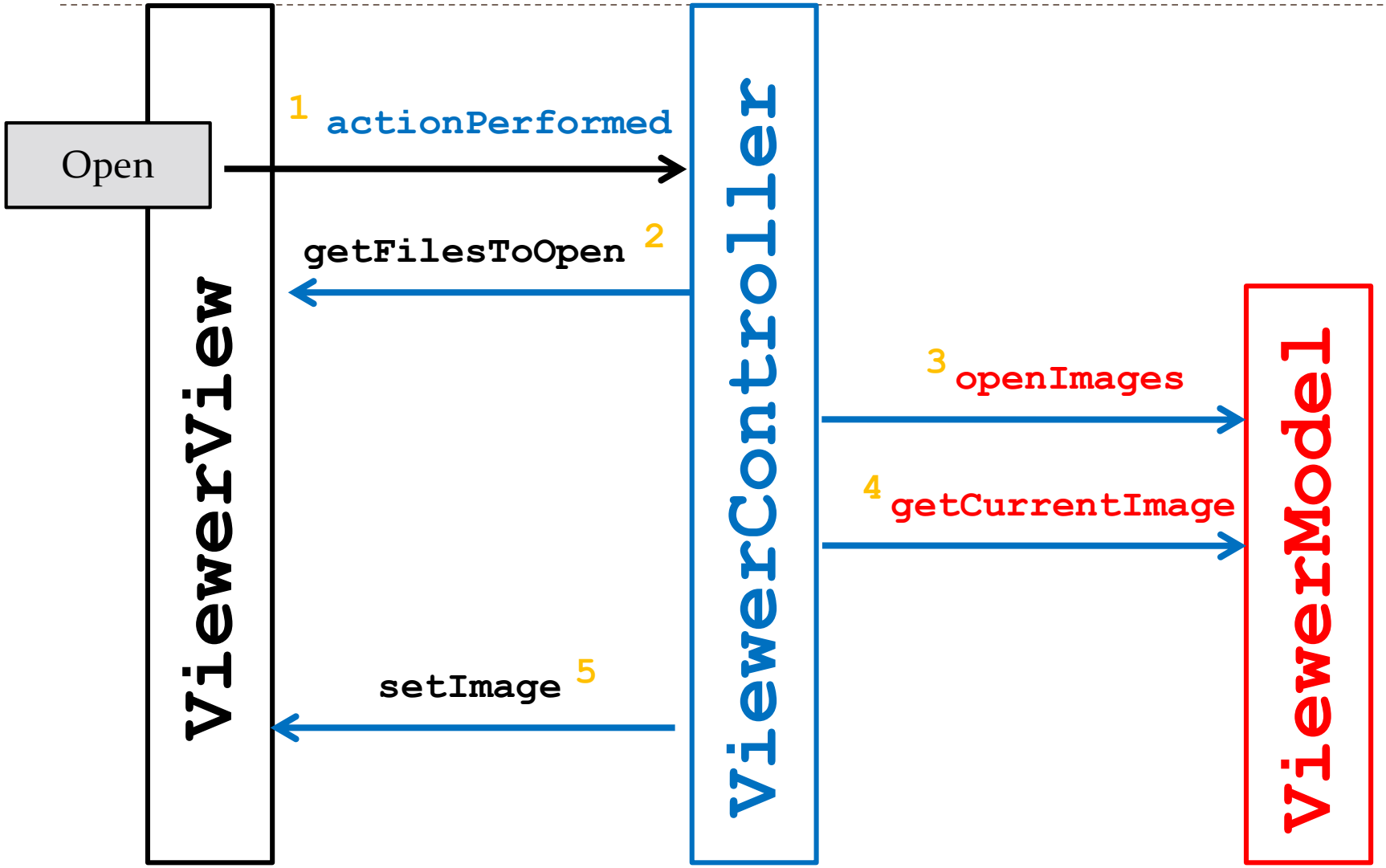
```
/**
 * The method that responds to events emitted by the view components.
 *
 * @param event
 *             an event emitted by the view
 *
 */
@Override
public void actionPerformed(ActionEvent event) {

}
```

```
@Override
public void actionPerformed(ActionEvent event) {
  String action = event.getActionCommand();









}
```

```
@Override
public void actionPerformed(ActionEvent event) {
  String action = event.getActionCommand();
  if (action.equals(ViewerView.OPEN)) {



  } else if (action.equals(ViewerView.CLEAR)) {



  } else if (action.equals(ViewerView.PREVIOUS)) {


  } else if (action.equals(ViewerView.NEXT)) {


  }
}
```

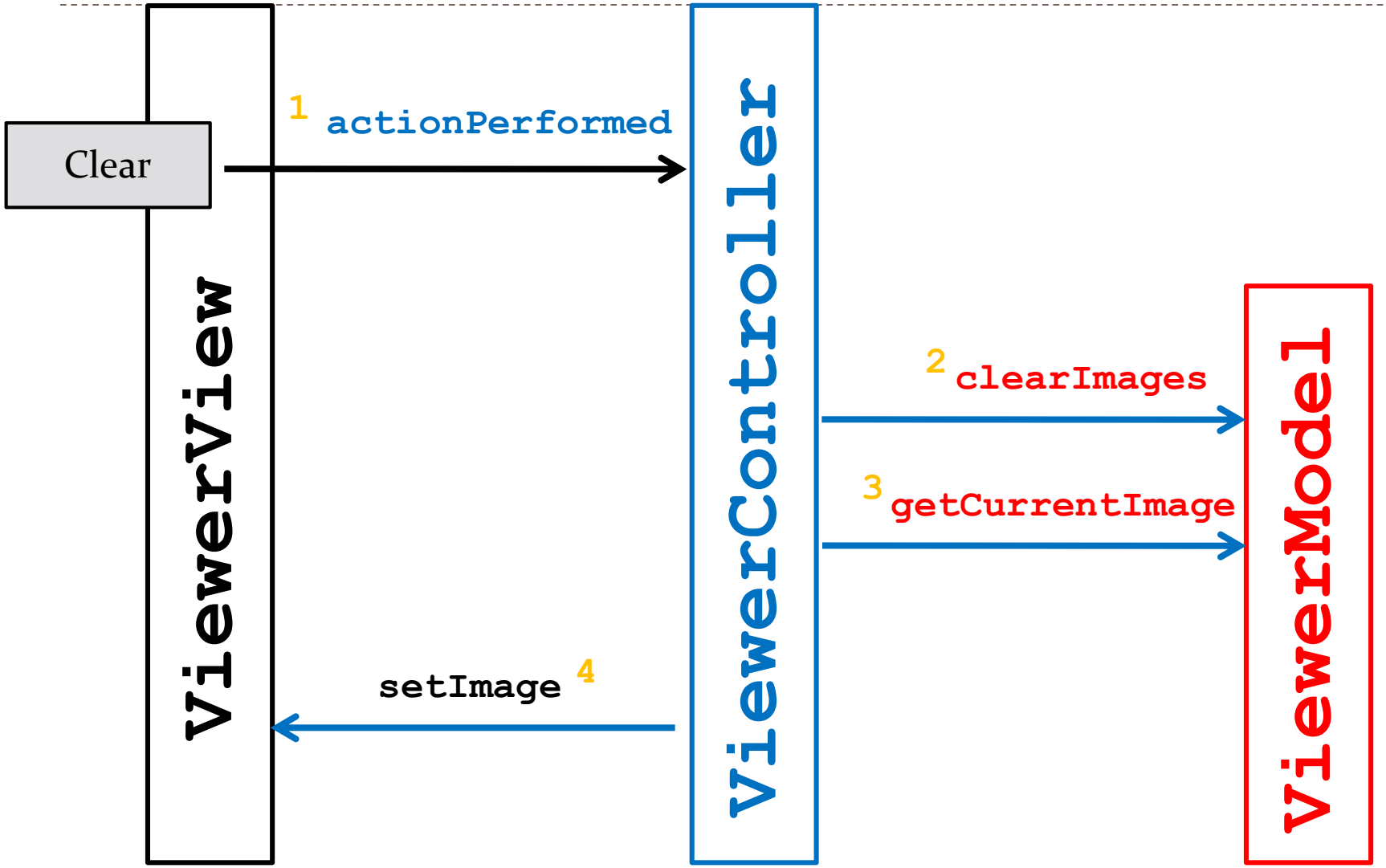
```
@Override
public void actionPerformed(ActionEvent event) {
  String action = event.getActionCommand();
  if (action.equals(ViewerView.OPEN)) {
    File[] files = this.view.getFilesToOpen();
    this.model.openImages(files);
    ImageIcon icon = this.model.getCurrentImage();
    this.view.setImage(icon);
  } else if (action.equals(ViewerView.CLEAR)) {



  } else if (action.equals(ViewerView.PREVIOUS)) {



  } else if (action.equals(ViewerView.NEXT)) {



  }
}
```

User Chooses Open Menu Item



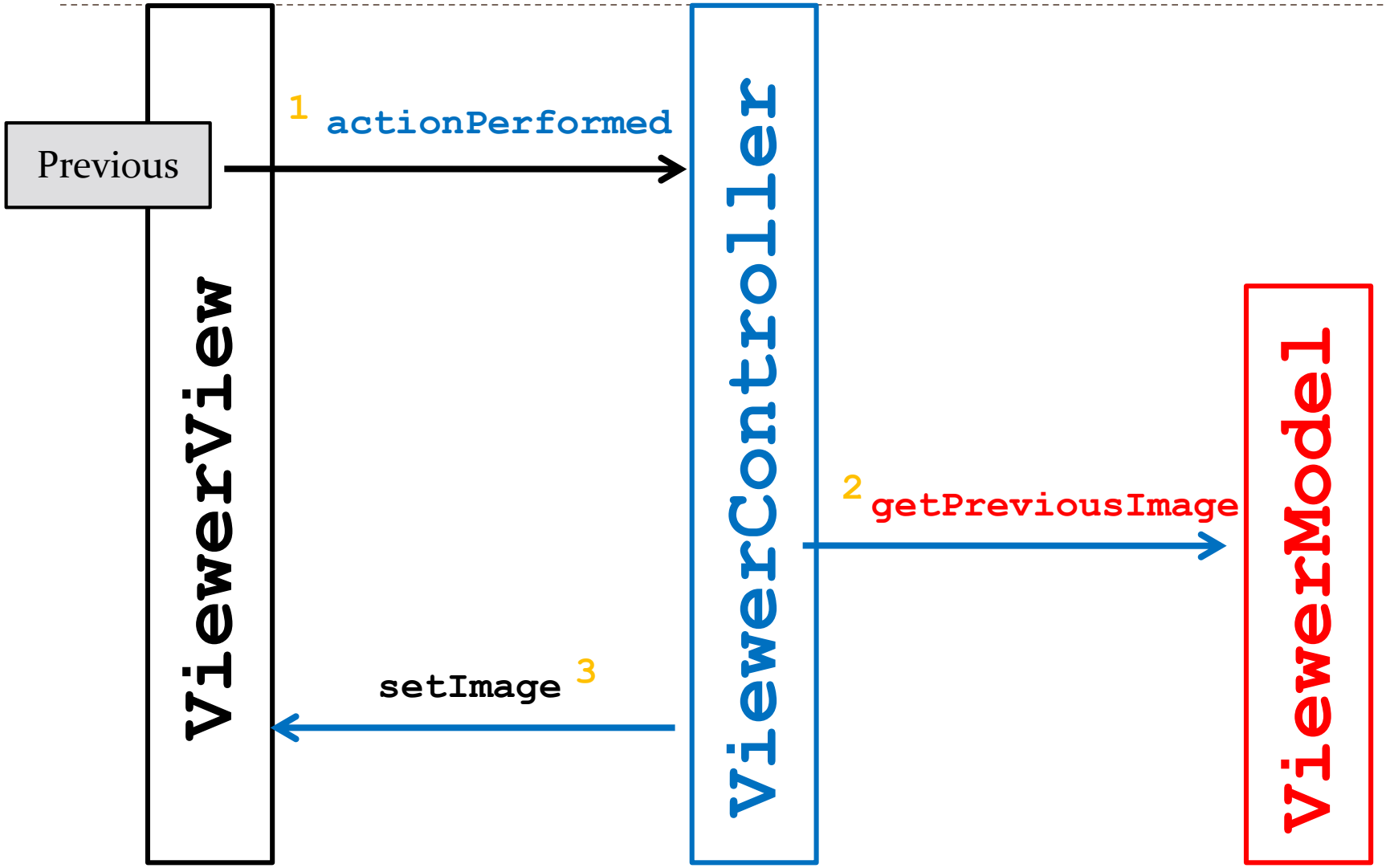
```
@Override
public void actionPerformed(ActionEvent event) {
  String action = event.getActionCommand();
  if (action.equals(ViewerView.OPEN)) {
    File[] files = this.view.getFilesToOpen();
    this.model.openImages(files);
    ImageIcon icon = this.model.getCurrentImage();
    this.view.setImage(icon);
  } else if (action.equals(ViewerView.CLEAR)) {
    this.model.clearImages();
    ImageIcon icon = this.model.getCurrentImage();
    this.view.setImage(icon);
  } else if (action.equals(ViewerView.PREVIOUS)) {


  } else if (action.equals(ViewerView.NEXT)) {


  }
}
```

User Chooses Clear Menu Item



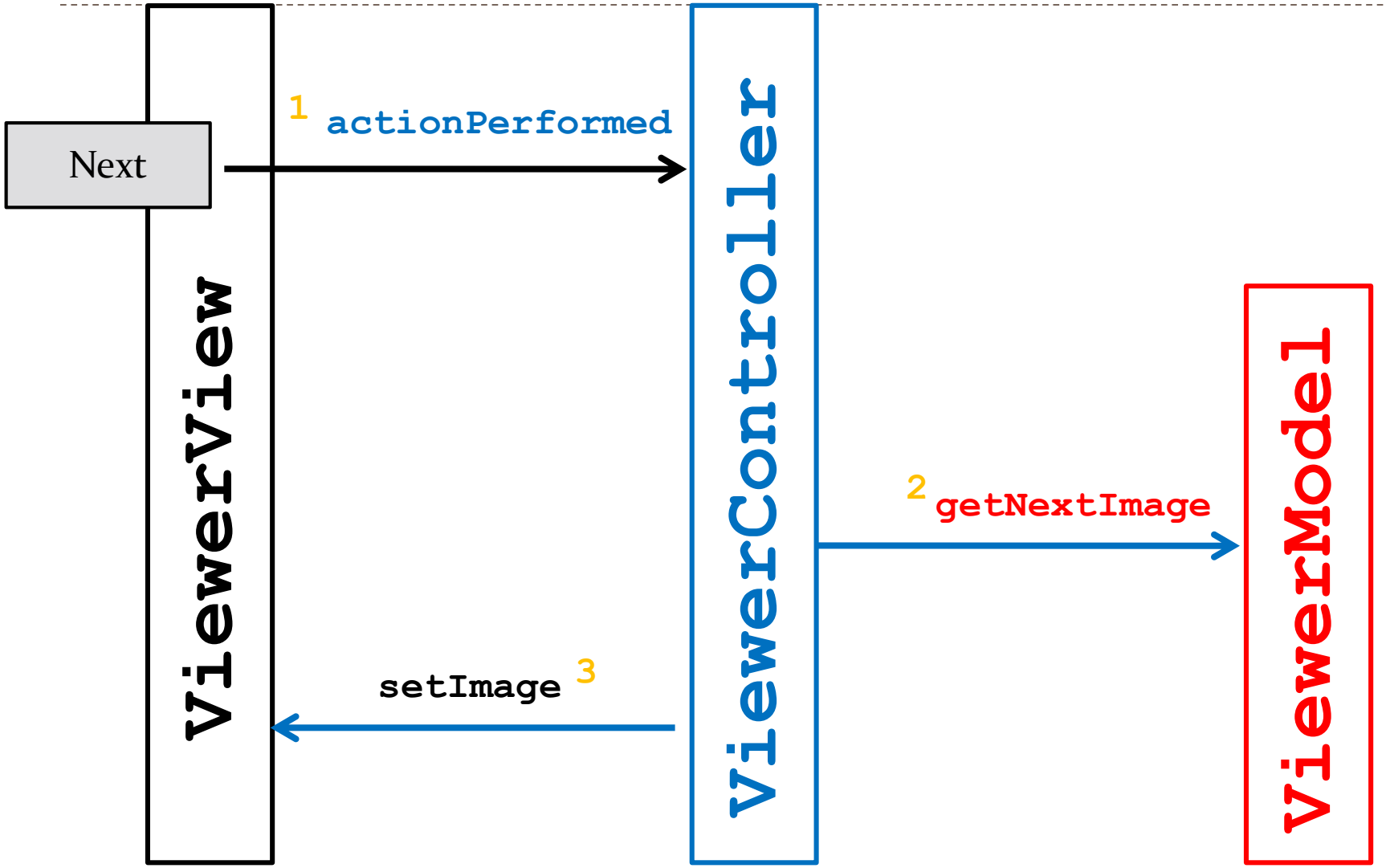
```
@Override
public void actionPerformed(ActionEvent event) {
  String action = event.getActionCommand();
  if (action.equals(ViewerView.OPEN)) {
    File[] files = this.view.getFilesToOpen();
    this.model.openImages(files);
    ImageIcon icon = this.model.getCurrentImage();
    this.view.setImage(icon);
  } else if (action.equals(ViewerView.CLEAR)) {
    this.model.clearImages();
    ImageIcon icon = this.model.getCurrentImage();
    this.view.setImage(icon);
  } else if (action.equals(ViewerView.PREVIOUS)) {
    ImageIcon icon = this.model.getPreviousImage();
    this.view.setImage(icon);
  } else if (action.equals(ViewerView.NEXT)) {



  }
}
```

User Presses Previous Button



```
@Override
public void actionPerformed(ActionEvent event) {
  String action = event.getActionCommand();
  if (action.equals(ViewerView.OPEN)) {
    File[] files = this.view.getFilesToOpen();
    this.model.openImages(files);
    ImageIcon icon = this.model.getCurrentImage();
    this.view.setImage(icon);
  } else if (action.equals(ViewerView.CLEAR)) {
    this.model.clearImages();
    ImageIcon icon = this.model.getCurrentImage();
    this.view.setImage(icon);
  } else if (action.equals(ViewerView.PREVIOUS)) {
    ImageIcon icon = this.model.getPreviousImage();
    this.view.setImage(icon);
  } else if (action.equals(ViewerView.NEXT)) {
    ImageIcon icon = this.model.getNextImage();
    this.view.setImage(icon);
  }
}
```

User Presses Next Button



The Image Viewer Application

- we need one more class that represents the image viewer application
  - has a **`main`** method
  - makes a **`ViewerModel`**
  - makes a **`ViewerController`**
  - makes a **`ViewerView`**
    - passes the controller to the view constructor so that the view knows where to send events
  - tells the controller about the model and the view
  - makes the view visible

```
public class ViewerApp {
  private ViewerApp() {

  }
  /**
   * The entry point to the image viewer application. Creates the model,
   * view, and controller, and makes the view visible.
   *
   * @param args not used
   */
  public static void main(String[] args) {
    ViewerModel model = new ViewerModel();
    ViewerController controller = new ViewerController();
    ViewerView view = new ViewerView(controller);
    controller.setModel(model);
    controller.setView(view);
    view.setVisible(true);
  }
}
```

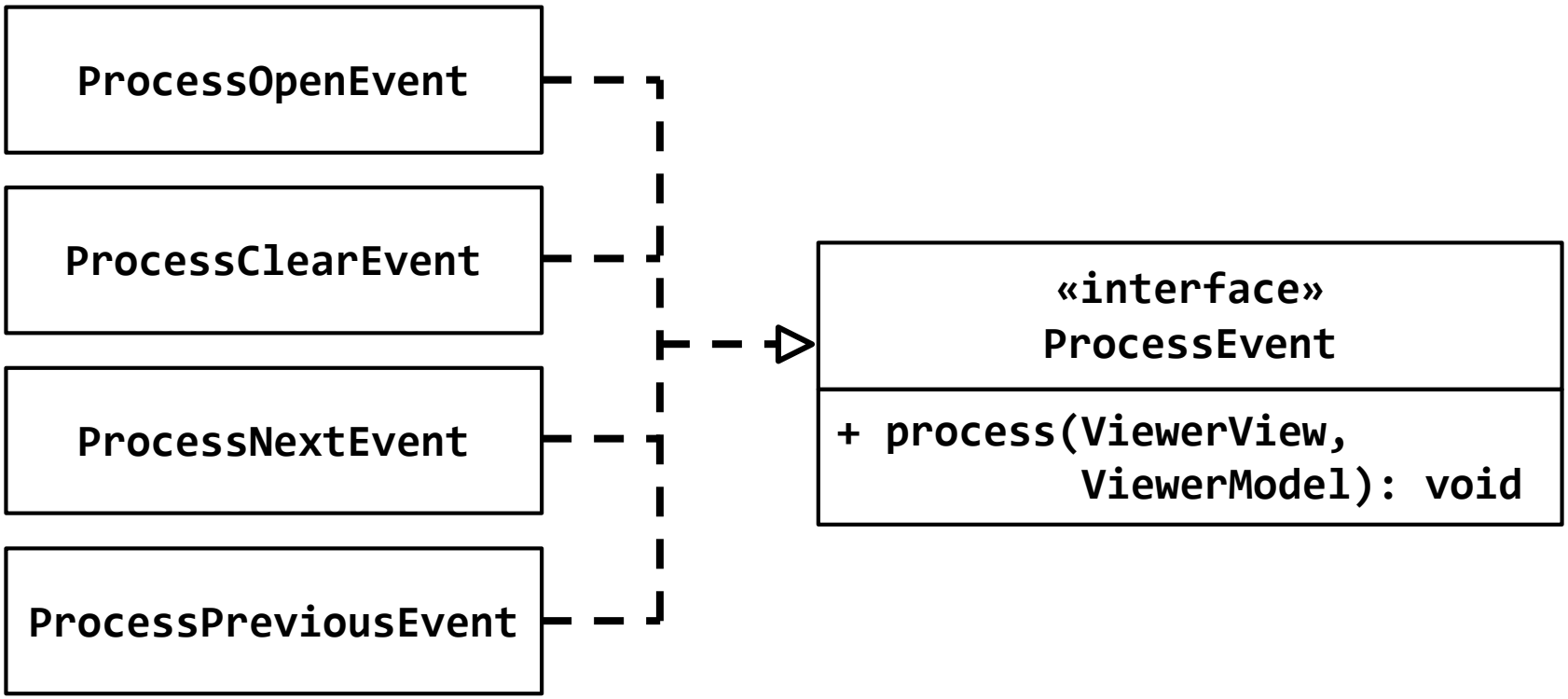
actionPerformed

- even with only 2 buttons and 2 menu items our **`actionPerformed`** method is unwieldy
  - imagine what would happen if you tried to implement a Controller this way for a big application with hundreds of buttons and menu items

Processing an event

- each part of the if statement in our current implementation of **`actionPerformed`** deals with processing of a different kind of event
  - "open" event
  - "clear" event
  - "next" event
  - "previous" event
- rather than one big **`actionPerformed`** method we can encapsulate the processing of each event in a separate object

Processing an event



```
public interface ProcessEvent {

    public void process(ViewerView view, ViewerModel model);

}
```

a method that encapsulates the processing of an event; every class that implements this interface must implement **process**

ProcessOpenEvent

```
import java.io.File;
import javax.swing.ImageIcon;

public class ProcessOpenEvent implements ProcessEvent {

    @Override
    public void process(ViewerView view, ViewerModel model) {
        File[] files = view.getFilesToOpen();
        model.openImages(files);
        ImageIcon icon = model.getCurrentImage();
        view.setImage(icon);
    }
}
```

code that was in `actionPerformed` now goes here

ProcessClearEvent

```
import javax.swing.ImageIcon;

public class ProcessClearEvent implements ProcessEvent {

    @Override
    public void process(ViewerView view, ViewerModel model) {
        model.clearImages();
        ImageIcon icon = model.getCurrentImage();
        view.setImage(icon);
    }

}
```

code that was in `actionPerformed` now goes here

ProcessNextEvent

```
import javax.swing.ImageIcon;


public class ProcessNextEvent implements ProcessEvent {

    @Override
    public void process(ViewerView view, ViewerModel model) {
        ImageIcon icon = model.getNextImage();
        view.setImage(icon);
    }

}
```

code that was in `actionPerformed` now goes here

```
import javax.swing.ImageIcon;

public class ProcessPreviousEvent implements ProcessEvent {

    @Override
    public void process(ViewerView view, ViewerModel model) {
        ImageIcon icon = model.getPreviousImage();
        view.setImage(icon);
    }

}
```

code that was in `actionPerformed` now goes here

Viewer Controller

- now that we've moved all of the event processing code out of **`actionPerformed`** and into separate classes, what does **`actionPerformed`** look like?

```
public class ViewerController2 implements ActionListener {

  private ViewerModel model;
  private ViewerView view;

  // constructor, setView, and setModel all remain the same as ViewerController

  @Override
  public void actionPerformed(ActionEvent event) {
    String action = event.getActionCommand();
    ProcessEvent pe = // what goes here? how do we get the right ProcessEvent
                      // object?
    pe.process();
  }

}
```

Viewer Controller

- the problem we have is that we need to get or create a **ProcessEvent** object that can handle the event whose action command string is given by **action**
- a solution is to call a method that knows how to create the appropriate **ProcessEvent** object given an action command string

```
public class ProcessEventFactory {

    public static ProcessEvent create(String action) {
        switch (action) {
        case ViewerView.OPEN:
            return new ProcessOpenEvent();
        case ViewerView.CLEAR:
            return new ProcessClearEvent();
        case ViewerView.NEXT:
            return new ProcessNextEvent();
        case ViewerView.PREVIOUS:
            return new ProcessPreviousEvent();
        default:
            throw new IllegalArgumentException();
        }
    }
}
```

static factory method:
A static method that creates and returns a new object; in this case, the method returns objects of different types depending on the value of the string **action**

```
public class ViewerController2 implements ActionListener {

    private ViewerModel model;
    private ViewerView view;

    // constructor, setView, and setModel all remain the same as ViewerController

    @Override
    public void actionPerformed(ActionEvent event) {
        String action = event.getActionCommand();
        ProcessEvent pe = ProcessEventFactory.create(action);
        pe.process(this.view, this.model);
    }

}
```

calls the factory method to get an object to handle the event; notice that we never see the actual type of **pe**

What have we gained?

- moved code that processes an event into separate classes
  - easier to debug, maintain, and extend
  - possible to handle new events by creating new subclasses of **ProcessEvent**
- removed an if statement from **actionPerformed**
  - but replaced it with a switch statement in the factory class
- handling new events requires modifying the factory class

A better factory

- we can build a better factory class
- the key insight is to realize that the factory *maps* a **String** onto a **ProcessEvent**
  - maybe we should use a **Map<String, ProcessEvent>** object to store **ProcessEvent** objects
  - it is probably a bad idea to have multiple factories (and thus, multiple possibly different mappings of **String**s to **ProcessEvent**s)
    - but we have a solution for that, too

```
import java.util.HashMap;
import java.util.Map;

public class ProcessEventFactory {

   private Map<String, ProcessEvent> map;

   public static final ProcessEventFactory instance = new ProcessEventFactory();

   private ProcessEventFactory() {
      this.map = new HashMap<String, ProcessEvent>();
      this.map.put(ViewerView.OPEN, new ProcessOpenEvent());
      this.map.put(ViewerView.CLEAR, new ProcessClearEvent());
      this.map.put(ViewerView.NEXT, new ProcessNextEvent());
      this.map.put(ViewerView.PREVIOUS, new ProcessPreviousEvent());
   }
```

map **String**s to **ProcessEvent**s

singleton

```
public ProcessEvent getProcessEvent(String event) {
    ProcessEvent pe = this.map.get(event);
    if (pe == null) {
        throw new IllegalArgumentException("cannot process " + event);
    }
    return pe;
}
```

a method that returns an appropriate **ProcessEvent** object by retrieving it from the map

```
public void register(String event, ProcessEvent p) {
   this.map.put(event, p);
}
```

a method that allows a client to add **ProcessEvent** objects to handle new types of events

```
public class ViewerController2 implements ActionListener {

    private ViewerModel model;
    private ViewerView view;

    // constructor, setView, and setModel all remain the same as ViewerController

    @Override
    public void actionPerformed(ActionEvent event) {
        String action = event.getActionCommand();
        ProcessEvent pe = ProcessEventFactory.instance.getProcessEvent(action);
        pe.process(this.view, this.model);
    }

}
```

uses the singleton to get an object to handle the event; notice that we never see the actual type of **pe**

Recursion

notes Chapter 8

Printing n of Something

- suppose you want to implement a method that prints out n copies of a string

```
public static void printIt(String s, int n) {
  for(int i = 0; i < n; i++) {
    System.out.print(s);
  }
}
```

A Different Solution

- alternatively we can use the following algorithm:
  1. if n == 0 done, otherwise
     I. print the string once
     II. print the string (n – 1) more times

```
public static void printItToo(String s, int n) {
  if (n == 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);    // method invokes itself
  }
}
```

Recursion

- a method that calls itself is called a *recursive* method
- a recursive method solves a problem by repeatedly reducing the problem so that a base case can be reached

```
printItToo("*", 5)
*printItToo ("*", 4)
**printItToo ("*", 3)
***printItToo ("*", 2)
****printItToo ("*", 1)
*****printItToo ("*", 0) base case
*****
```

Notice that the number of times the string is printed decreases after each recursive call to printIt

Notice that the base case is eventually reached.

Base cases

- a base case is a version of the problem that the method is trying to solve for which the answer is known
  - because the answer is known no further recursion is required
- for many recursive algorithms, there is one obvious base case which is the smallest version of the problem for which the answer is known

Infinite Recursion

- if the base case(s) is missing, or never reached, a recursive method will run forever (or until the computer runs out of resources)

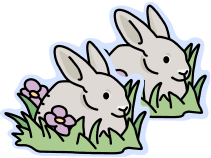
```
public static void printItForever(String s, int n) {
  // missing base case; infinite recursion
  System.out.print(s);
  printItForever(s, n - 1);
}

  printItForever("*", 1)
  * printItForever("*", 0)
  ** printItForever("*", -1)
  *** printItForever("*", -2) ..........
```

Climbing a Flight of n Stairs

- not Java

```
/**
  * method to climb n stairs
  */
climb(n) :
if n == 0
  done
else
  step up 1 stair
  climb(n - 1);
end
```

Rabbits



Month 0: 1 pair

0 additional pairs

Month 1: first pair makes another pair

1 additional pair

Month 2: each pair makes another pair; oldest pair dies

1 additional pair

Month 3: each pair makes another pair; oldest pair dies

2 additional pairs

Fibonacci Numbers

- the sequence of additional pairs
  - `0, 1, 1, 2, 3, 5, 8, 13, ...`

  are called Fibonacci numbers

- base cases
  - `F(0) = 0`
  - `F(1) = 1`
- recursive definition
  - `F(n) = F(n - 1) + F(n - 2)`

Recursive Methods & Return Values

- a recursive method can return a value
- example: compute the nth Fibonacci number

```
public static int fibonacci(int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1) {
    return 1;
  }
  else {
   int f = fibonacci(n - 1) + fibonacci(n - 2);
   return f;
  }
}
```

Recursive Methods & Return Values

- write a recursive method that multiplies two positive integer values (i.e., both values are strictly greater than zero)

- observation: $m \times n$ means add m n's together
  - in other words, you can view multiplication as recursive addition

Recursive Methods & Return Values

- not Java:

```
/**
 * Computes m * n
 */
multiply(m, n) :
if m == 1
    return n
else
    return n + multiply(m - 1, n)
```

```
public static int multiply(int m, int n) {
  if (m == 1) {
    return n;
  }
  return n + multiply(m - 1, n);
}
```

Recursive Methods & Return Values

- example: write a recursive method **`countZeros`** that counts the number of zeros in an integer number **`n`**
  - **`10305060700002L`** has 8 zeros

- trick: examine the following sequence of numbers
  1. `10305060700002`
  2. `1030506070000`
  3. `103050607000`
  4. `10305060700`
  5. `103050607`
  6. `1030506 ...`

Recursive Methods & Return Values

- not Java:

```
/**
 * Counts the number of zeros in an integer n
 */
countZeros(n) :
if the last digit in n is a zero
    return 1 + countZeros(n / 10)
else
    return countZeros(n / 10)
```

- don't forget to establish the base case(s)
  - when should the recursion stop? when you reach a single digit (not zero digits; you never reach zero digits!)
    - base case #1 : `n == 0`
      - `return 1`
    - base case #2 : `n != 0 && n < 10`
      - `return 0`

```
public static int countZeros(long n) {

  if(n == 0L) {  // base case 1
    return 1;
  }
  else if(n < 10L) {  // base case 2
    return 0;
  }

  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero) {
    return 1 + countZeros(m);
  }
  else {
    return countZeros(m);
  }
}
```

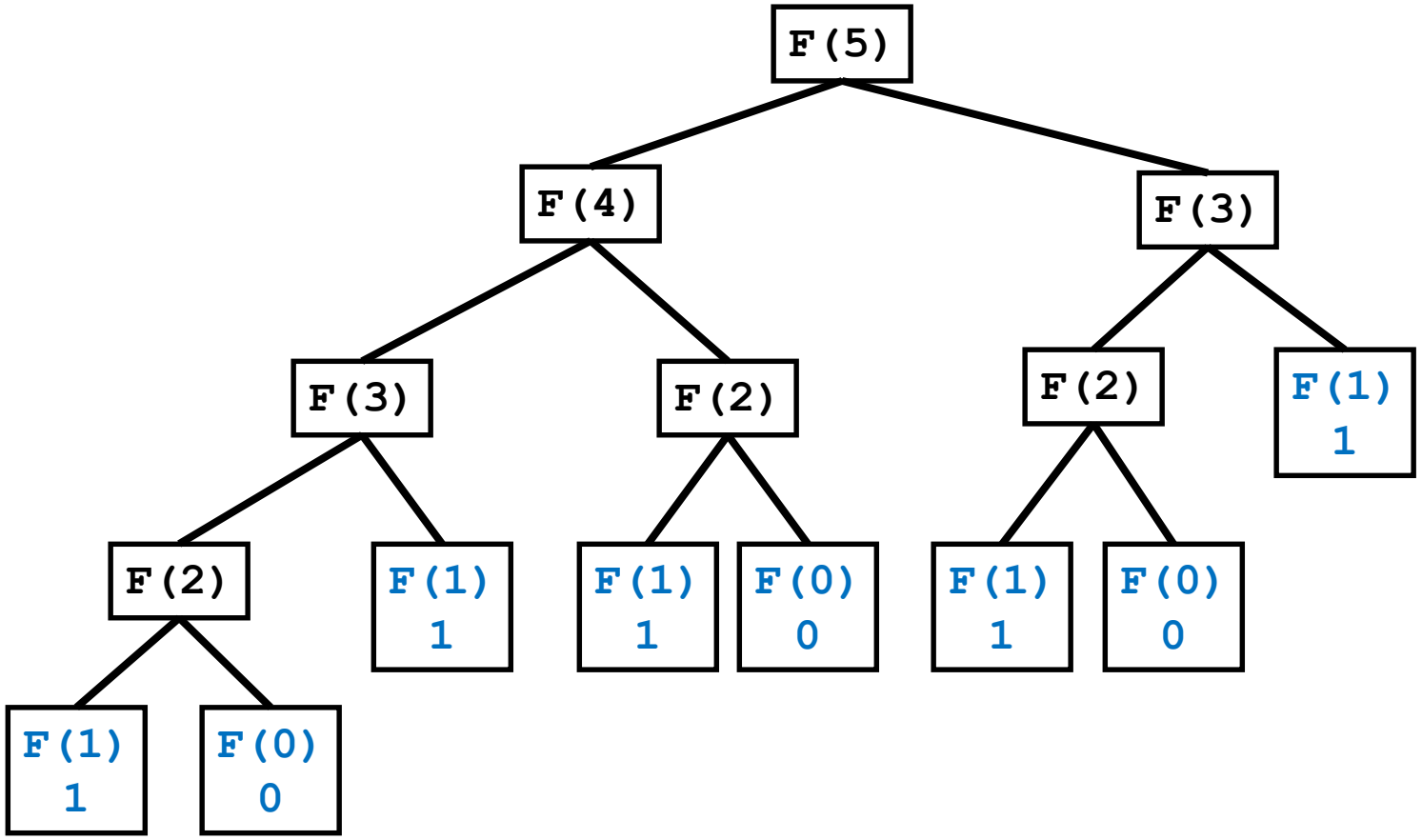
countZeros Call Stack

`callZeros( 800410L )`

last in first out

`callZeros( 8L )`	`0`
`callZeros( 80L )`	`1 + 0`
`callZeros( 800L )`	`1 + 1 + 0`
`callZeros( 8004L )`	`0 + 1 + 1 + 0`
`callZeros( 80041L )`	`0 + 0 + 1 + 1 + 0`
`callZeros( 800410L )`	`1 + 0 + 0 + 1 + 1 + 0`
	`= 3`

Fibonacci Call Tree



Compute Powers of 10

- write a recursive method that computes 10^n for any integer value **n**
- recall:
  - $10^0 = 1$
  - $10^n = 10 * 10^{n-1}$
  - $10^{-n} = 1 / 10^n$

```
public static double powerOf10(int n) {
  if (n == 0) {
    // base case
    return 1.0;
  }
  else if (n > 0) {
    // recursive call for positive n
    return 10.0 * powerOf10(n - 1);
  }
  else {
    // recursive call for negative n
    return 1.0 / powerOf10(-n);
  }
}
```

Fibonacci Numbers

- the sequence of additional pairs
  - `0, 1, 1, 2, 3, 5, 8, 13, ...`

  are called Fibonacci numbers

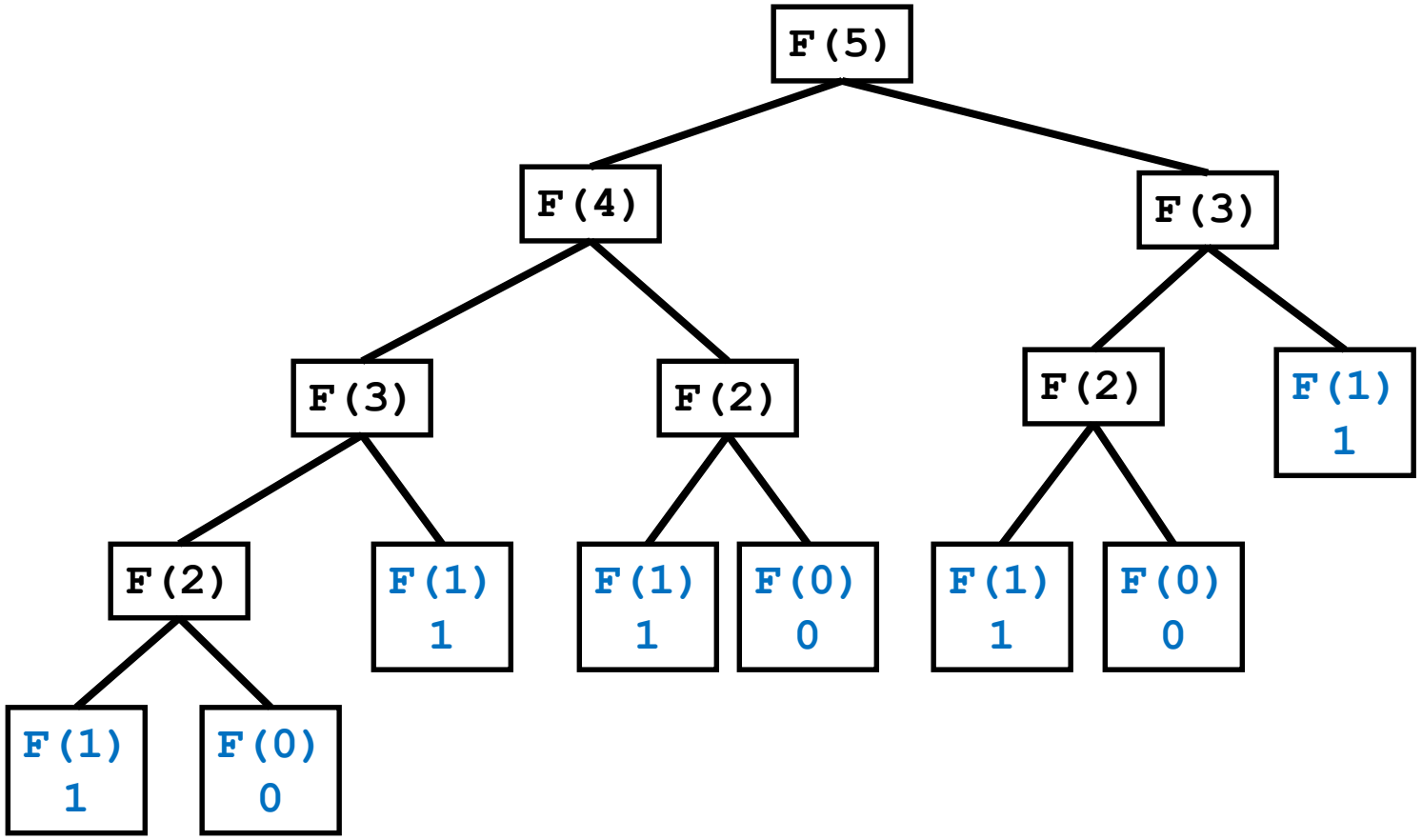
- base cases
  - `F(0) = 0`
  - `F(1) = 1`
- recursive definition
  - `F(n) = F(n - 1) + F(n - 2)`

Recursive Methods & Return Values

- a recursive method can return a value
- example: compute the nth Fibonacci number

```
public static int fibonacci(int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1) {
    return 1;
  }
  else {
   int f = fibonacci(n - 1) + fibonacci(n - 2);
   return f;
  }
}
```

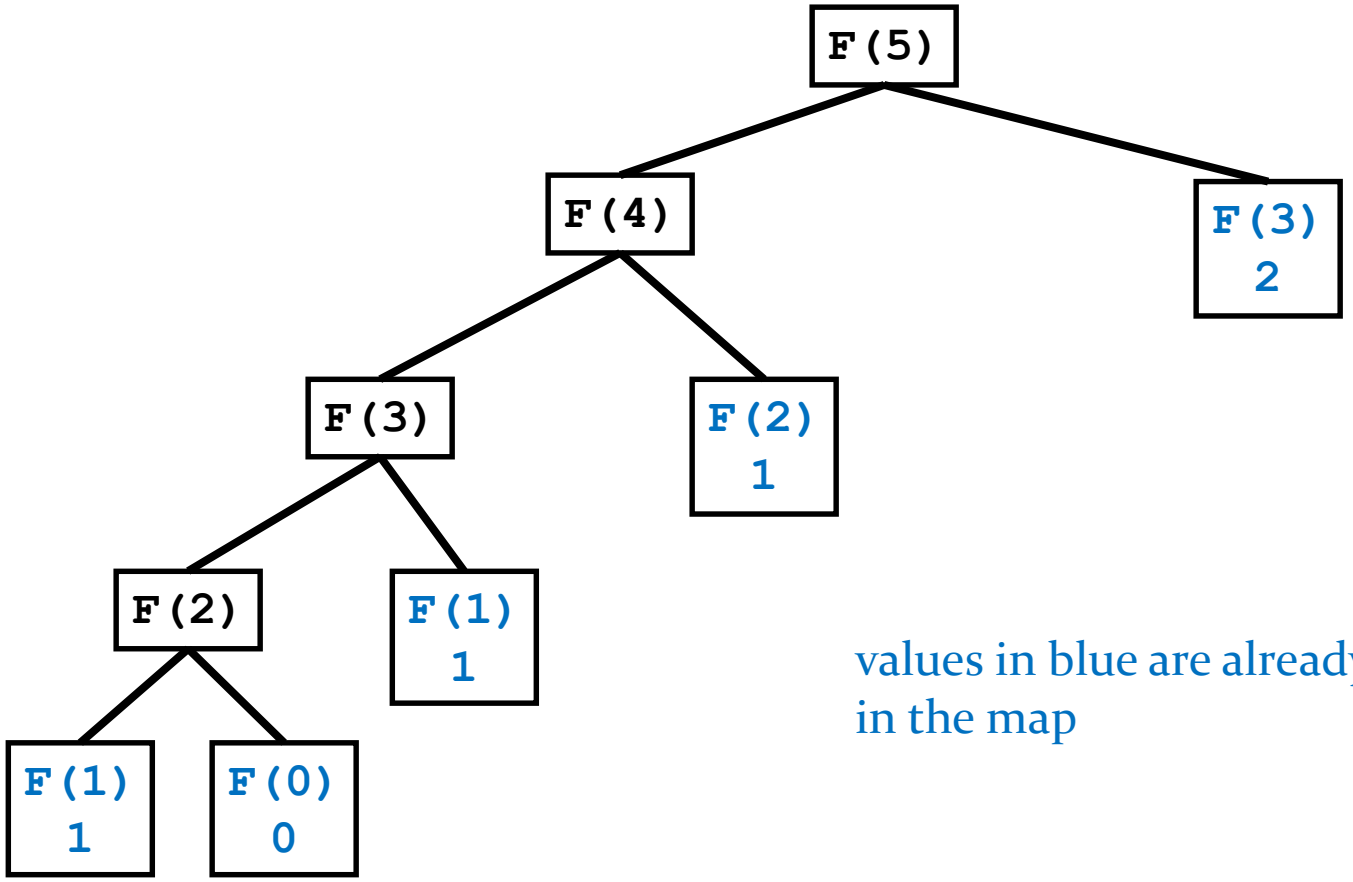
Fibonacci Call Tree



A Better Recursive Fibonacci

```
public class Fibonacci {
  private static Map<Integer, Long> values = new HashMap<Integer, Long>();
  static {
    Fibonacci.values.put(0, (long) 0);
    Fibonacci.values.put(1, (long) 1);
  }

  public static long getValue(int n) {
    Long value = Fibonacci.values.get(n);
    if (value != null) {
      return value;
    }
    value = Fibonacci.getValue(n - 1) + Fibonacci.getValue(n - 2);
    Fibonacci.values.put(n, value);
    return value;
  }
}
```

Better Fibonacci Call Tree



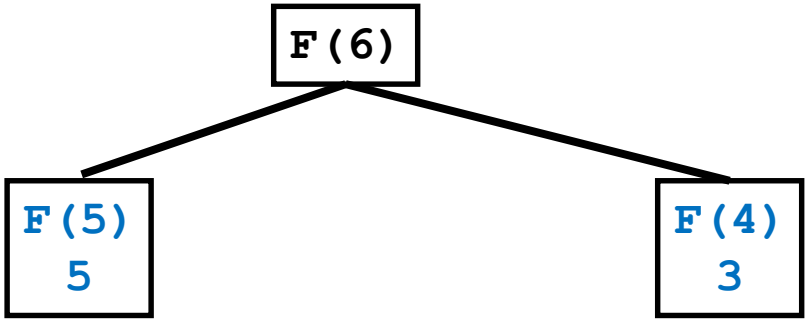
values in blue are already stored in the map

A Better Recursive Fibonacci

- because the map is static subsequent calls to **Fibonacci.*getValue*(int)** can use the values already computed and stored in the map

Better Fibonacci Call Tree

- assuming the client has already invoked **Fibonacci.*getValue*(5)**



values in blue are already stored in the map

Compute Powers of 10

- write a recursive method that computes 10^n for any integer value **n**
- recall:
  - $10^n = 1 / 10^{-n}$ if **n** < 0
  - $10^0 = 1$
  - $10^n = 10 * 10^{n-1}$

```
public static double powerOf10(int n) {
  if (n < 0) {
    return 1.0 / powerOf10(-n);
  }
  else if (n == 0) {
    return 1.0;
  }
  return n * powerOf10(n - 1);
}
```

A Better Powers of 10

- recall:
  - $10^n = 1 / 10^{-n}$ if **n** < 0
  - $10^0 = 1$
  - $10^n = 10 * 10^{n-1}$ if **n** is odd
  - $10^n = 10^{n/2} * 10^{n/2}$ if **n** is even

```
public static double powerOf10(int n) {
  if (n < 0) {
    return 1.0 / powerOf10(-n);
  }
  else if (n == 0) {
    return 1.0;
  }
  else if (n % 2 == 1) {
    return 10 * powerOf10(n - 1);
  }
  double value = powerOf10(n / 2);
  return value * value;
}
```