Inheritance (Part 4)

Abstract Classes

Polymorphism

- inheritance allows you to define a base class that has fields and methods
  - classes derived from the base class can use the public and protected base class fields and methods
- polymorphism allows the implementer to change the behaviour of the derived class methods

```
// client code
public void print(Dog d) {
  System.out.println( d.toString() );
}

// later on...
Dog           fido = new Dog();
CockerSpaniel lady = new CockerSpaniel();
Mix           mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
```

```
Dog toString
CockerSpaniel toString
Mix toString
```

- notice that **`fido`**, **`lady`**, and **`mutt`** were declared as **`Dog`**, **`CockerSpaniel`**, and **`Mutt`**
- what if we change the declared type of **`fido`**, **`lady`**, and **`mutt`** ?

```
// client code
public void print(Dog d) {
  System.out.println( d.toString() );
}

// later on...
Dog           fido = new Dog();
Dog           lady = new CockerSpaniel();
Dog           mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
```

```
Dog toString
CockerSpaniel toString
Mix toString
```

- what if we change the **`print`** method parameter type to **`Object`** ?

```
// client code
public void print(Object obj) {
  System.out.println( obj.toString() );
}

// later on...
Dog           fido = new Dog();
Dog           lady = new CockerSpaniel();
Dog           mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
this.print(new Date());
```

```
Dog toString
CockerSpaniel toString
Mix toString
Date toString
```

Late Binding

- polymorphism requires *late binding* of the method name to the method definition
  - late binding means that the method definition is determined at run-time

non-static method

```
obj.toString()
```

run-time type of the instance `obj`

Declared vs Run-time type

```
Dog lady = new CockerSpaniel();
```

declared type

run-time or actual type

- the declared type of an instance determines what methods can be used

```
Dog lady = new CockerSpaniel();
```

  - the name `lady` can only be used to call methods in `Dog`
  - `lady.someCockerSpanielMethod()` won't compile

Dynamic dispatch

- the actual type of the instance determines what definition is used when the method is called

```
Dog lady = new CockerSpaniel();
```

  - `lady.toString()` uses the `CockerSpaniel` definition of `toString`

- selecting which version of a polymorphic method to use at run-time is called *dynamic dispatch*

Abstract classes

Abstract Classes

- sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - e.g. you might want to know what a **`Dog`**'s bark sounds like but the sound of the bark depends on the breed of the dog
    - you want to add the method bark to **`Dog`** but only the subclasses of **`Dog`** can implement **`bark`**

Abstract Classes

- sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - e.g. you might want to know the breed of a **`Dog`** but only the subclasses have information about the breed
    - you want to add the method **`getBreed`** to **`Dog`** but only the subclasses of **`Dog`** can implement **`getBreed`**

- if the base class has methods that only subclasses can define *and* the base class has fields common to all subclasses then the base class should be abstract
  - if you have a base class that just has methods that it cannot implement then you probably want an interface
- abstract :
    - (dictionary definition) existing only in the mind

- in Java an abstract class is a class that you cannot make instances of
  - e.g. http://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html

- an abstract class provides a partial definition of a class
  - the "partial definition" contains everything that is common to all of the subclasses
  - the subclasses complete the definition

- an abstract class can define fields and methods
  - subclasses *inherit* these
- an abstract class can define constructors
  - subclasses *must call* these
- an abstract class can declare abstract methods
  - subclasses *must define* these (unless the subclass is also abstract)

Abstract Methods

- an abstract base class can declare, *but not define*, zero or more abstract methods

```
public abstract class Dog
{
  // fields, ctors, regular methods

  public abstract String getBreed();
}
```

- the base class is saying "all **Dog**s can provide a **String** describing the breed, but only the subclasses know enough to implement the method"

Abstract Methods

- the non-abstract subclasses must provide definitions for all abstract methods
  - consider `getBreed` in `Mix`

```
public class Mix extends Dog
{ // stuff from before...

  @Override
  public String getBreed() {
    if(this.breeds.isEmpty()) {
      return "mix of unknown breeds";
    }
    StringBuffer b = new StringBuffer();
    b.append("mix of");
    for(String breed : this.breeds) {
      b.append(" " + breed);
    }
    return b.toString();
}
```

PureBreed

- a purebreed dog is a dog with a single breed
  - one `String` field to store the breed
- note that the breed is determined by the subclasses
  - the class `PureBreed` cannot give the `breed` field a value
  - but it can implement the method `getBreed`
- the class `PureBreed` defines an field common to all subclasses and it needs the subclass to inform it of the actual breed
  - `PureBreed` is also an abstract class

```
public abstract class PureBreed extends Dog
{
  private String breed;

  public PureBreed(String breed) {
    super();
    this.breed = breed;
  }

  public PureBreed(String breed, int size, int energy) {
    super(size, energy);
    this.breed = breed;
  }
  @Override public String getBreed()
  {
    return this.breed;
  }

}
```

Subclasses of PureBreed

- the subclasses of **`PureBreed`** are responsible for setting the breed
  - consider **`Komondor`**

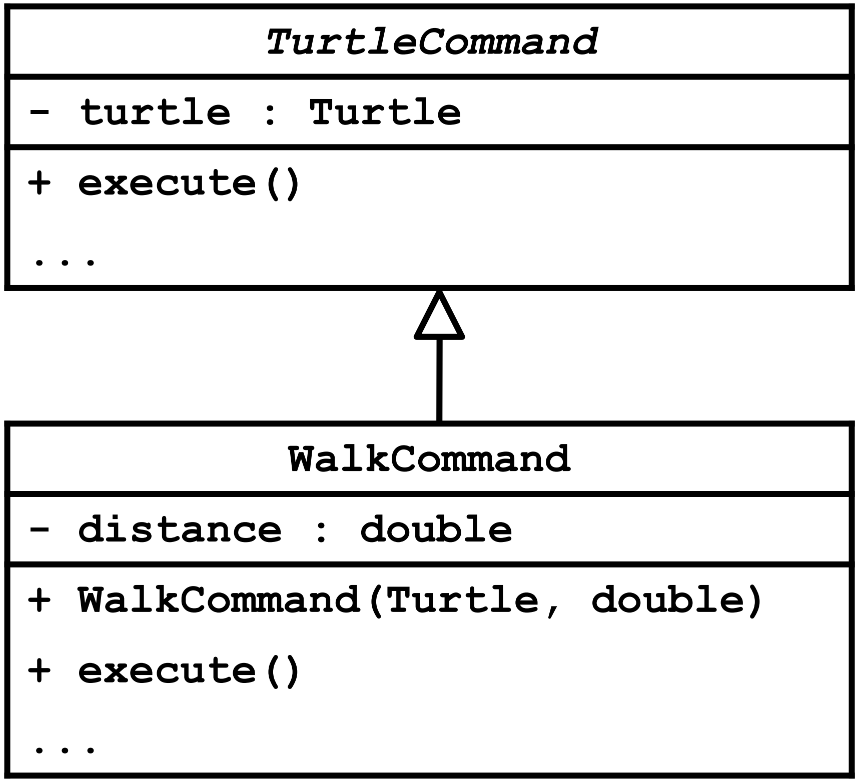
Komondor

```
public class Komondor extends PureBreed
{
  private final String BREED = "komondor";

  public Komondor() {
    super(BREED);
  }

  public Komondor(int size, int energy) {
    super(BREED, size, energy);
  }

  // other Komondor methods...
}
```

Another example: TurtleCommand

- from Lab 5:
  - every **`TurtleCommand`** object has a reference to a **`Turtle`**
    - the turtle which is affected by the command
  - every **`TurtleCommand`** object has an **`execute()`** method
    - causes the turtle to execute the command

- **`TurtleCommand`** cannot implement **`execute()`** because it doesn’t know about all of the different kinds of commands
  - **`TurtleCommand`** must be abstract



TurtleCommand

- turtle : Turtle

+ execute()

...

WalkCommand

- distance : double

+ WalkCommand(Turtle, double)

+ execute()

...

- class name in italics for abstract classes
- the Turtle affected by the command

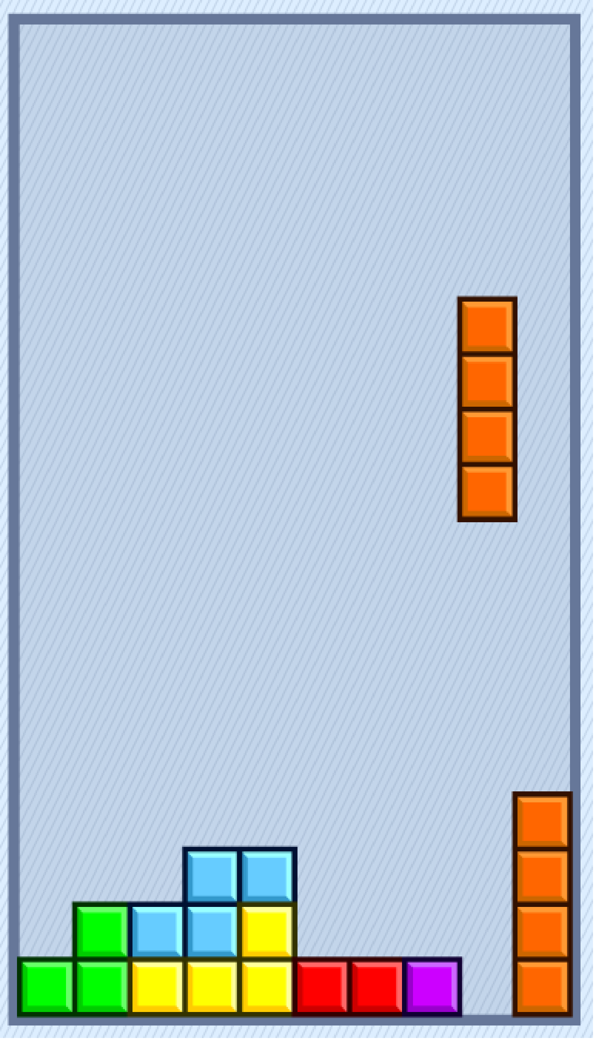
TurtleCommand is abstract because we can't implement **execute()**

- constructor sets the distance to walk
- concrete implementation of **execute()**

http://www.eecs.yorku.ca/course_archive/2016-17/W/2030/labs/lab5/lab5.html

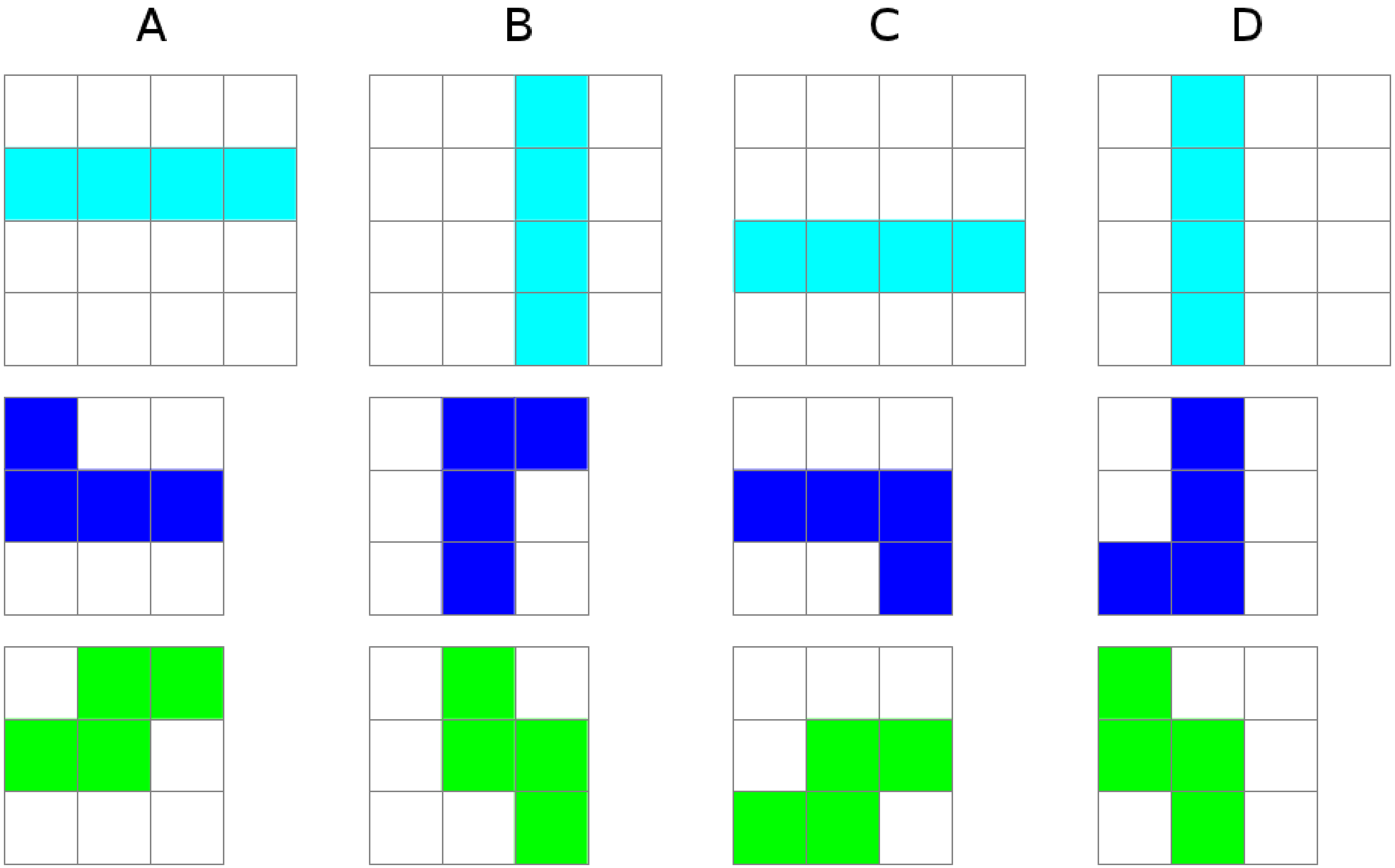
Another example: Tetris

- played with 7 standard blocks called tetriminoes
- blocks drop from the top
- player can move blocks left, right, and down
- player can spin blocks left and right



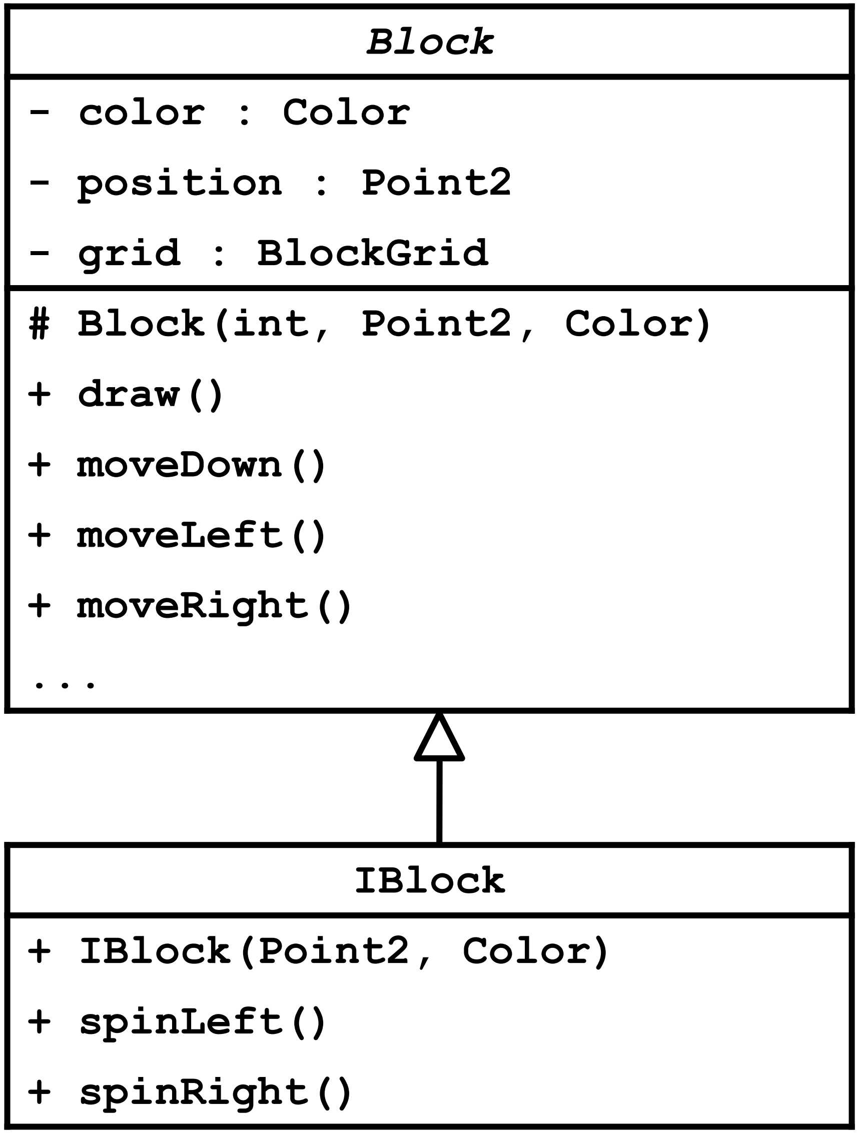
Tetriminoes

- spinning the I, J, and S blocks



Tetriminoes

- features common to all tetriminoes
  - has-a color
  - has-a shape
  - has-a position
  - draw
  - move left, right, and down
- features unique to each kind of tetrimino
  - the actual shape
  - spin left and right



```
Block
- color : Color
- position : Point2
- grid : BlockGrid
# Block(int, Point2, Color)
+ draw()
+ moveDown()
+ moveLeft()
+ moveRight()
...

IBlock
+ IBlock(Point2, Color)
+ spinLeft()
+ spinRight()
```

- class name in italics for abstract classes
- a 2D point
- a grid object that stores the shape
- protected constructor

`Block` is abstract because we can't define the shape of a generic block

- constructor defines the shape
- methods modify the shape to produce the rotated version of the block

http://www.eecs.yorku.ca/course_archive/2016-17/F/2030/labs/lab4/lab4.html

Inheritance (Part 5)

Static Features; Interfaces

Static Fields and Inheritance

- static fields behave the same as non-static fields in inheritance
  - public and protected static fields are inherited by subclasses, and subclasses can access them directly by name
  - private static fields are not inherited and cannot be accessed directly by name
    - but they can be accessed/modified using public and protected methods

Static Fields and Inheritance

- the important thing to remember about static fields and inheritance
  - *there is only one copy of the static field shared among the declaring class and all subclasses*

- consider trying to count the number of `Dog` objects created by using a static counter

```java
// the wrong way to count the number of Dogs created
public abstract class Dog {
  // other fields...
  static protected int numCreated = 0;

  Dog() {
    // ...
    Dog.numCreated++;
  }

  public static int getNumberCreated() {
    return Dog.numCreated;
  }

  // other contructors, methods...
}
```

protected, not private, so that subclasses can modify it directly

```
// the wrong way to count the number of Dogs created
public class Mix extends Dog
{
  // fields...

  Mix()
  {
    super();
    Mix.numCreated++;
  }

  // other contructors, methods...
}
```

```
// too many dogs!

public class TooManyDogs
{
  public static void main(String[] args)
  {
    Mix mutt = new Mix();
    System.out.println( Mix.getNumberCreated() );
  }
}
```

prints 2

What Went Wrong?

- there is only one copy of the static field shared among the declaring class and all subclasses
  - **`Dog`** declared the static field
  - **`Dog`** increments the counter every time its constructor is called
  - **`Mix`** inherits *and shares* the single copy of the field
  - **`Mix`** constructor correctly calls the superclass constructor
    - which causes **`numCreated`** to be incremented by **`Dog`**
  - **`Mix`** constructor then incorrectly increments the counter

Counting Dogs and Mixes

- suppose you want to count the number of **`Dog`** instances and the number of **`Mix`** instances
  - **`Mix`** must also declare a static field to hold the count
    - somewhat confusingly, **`Mix`** can give the counter the same name as the counter declared by **`Dog`**

```
public class Mix extends Dog
{
  // other fields...
  private static int numCreated = 0;  // bad style; hides Dog.numCreated

  public Mix()
  {
    super();      // will increment Dog.numCreated
    // other Mix stuff...
    numCreated++; // will increment Mix.numCreated
  }

  // ...
```

Hiding Fields

- note that the **`Mix`** field **`numCreated`** has the same name as an field declared in a superclass
  - whenever **`numCreated`** is used in **`Mix`**, it is the **`Mix`** version of the field that is used

- if a subclass declares an field with the same name as a superclass field, we say that the subclass field hides the superclass field
  - considered bad style because it can make code hard to read and understand
    - should change **`numCreated`** to **`numMixCreated`** in **`Mix`**

Static Methods and Inheritance

- there is a significant difference between calling a static method and calling a non-static method when dealing with inheritance

- *there is no dynamic dispatch on static methods*
  - therefore, you cannot override a static method

```
public abstract class Dog {
  private static int numCreated = 0;
  public static int getNumCreated() {
    return Dog.numCreated;
  }
}
```

```
public class Mix {
  private static int numMixCreated = 0;
  public static int getNumCreated() {
    return Mix.numMixCreated;
  }
}
```

notice no @Override

```
public class Komondor {
  private static int numKomondorCreated = 0;
  public static int getNumCreated() {
    return Komondor.numKomondorCreated;
  }
}
```

notice no @Override

```
public class WrongCount {
  public static void main(String[] args) {
    Dog mutt = new Mix();
    Dog shaggy = new Komondor();
    System.out.println( mutt.getNumCreated() );
    System.out.println( shaggy.getNumCreated() );
    System.out.println( Mix.getNumCreated() );
    System.out.println( Komondor.getNumCreated() );
  }
}
```

Dog version

Dog version

Mix version

Komondor version

```
prints 2
       2
       1
       1
```

What's Going On?

- *there is no dynamic dispatch on static methods*

- because the declared type of **`mutt`** is **`Dog`**, it is the **`Dog`** version of **`getNumCreated`** that is called

- because the declared type of **`shaggy`** is **`Dog`**, it is the **`Dog`** version of **`getNumCreated`** that is called

Hiding Methods

- notice that **`Mix.getNumCreated`** and **`Komondor.getNumCreated`** work as expected
- if a subclass declares a static method with the same name as a superclass static method, we say that the subclass static method hides the superclass static method
  - *you cannot override a static method, you can only hide it*
  - hiding static methods is considered bad form because it makes code hard to read and understand

- the client code in **`WrongCount`** illustrates two cases of bad style, one by the client and one by the implementer of the **`Dog`** hierarchy
  1. the client should not have used an instance to call a static method
  2. the implementer should not have hidden the static method in **`Dog`**

Using superclass methods

Other Methods

- methods in a subclass will often need or want to call methods in the immediate superclass
  - a new method in the subclass can call any **`public`** or **`protected`** method in the superclass without using any special syntax
- a subclass can override a **`public`** or **`protected`** method in the superclass by declaring a method that has the same signature as the one in the superclass
  - a subclass method that overrides a superclass method can call the overridden superclass method using the **`super`** keyword

Dog equals

- we will assume that two **Dog**s are equal if their size and energy are the same

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if(obj != null && this.getClass() == obj.getClass())
  {
    Dog other = (Dog) obj;
    eq = this.getSize() == other.getSize() &&
         this.getEnergy() == other.getEnergy();
  }
  return eq;
}
```

Mix equals (version 1)

- two Mix instances are equal if their Dog subobjects are equal and they have the same breeds

```
@Override public boolean equals(Object obj)
{ // the hard way
  boolean eq = false;
  if(obj != null && this.getClass() == obj.getClass()) {
    Mix other = (Mix) obj;
    eq = this.getSize() == other.getSize() &&
         this.getEnergy() == other.getEnergy() &&
         this.breeds.size() == other.breeds.size() &&
         this.breeds.containsAll(other.breeds);
  }
  return eq;
}
```

subclass can call public method of the superclass

Mix equals (version 2)

- two Mix instances are equal if their Dog subobjects are equal and they have the same breeds
  - Dog equals already tests if two Dog instances are equal
  - Mix equals can call Dog equals to test if the Dog subobjects are equal, and then test if the breeds are equal
- also notice that Dog equals already checks that the Object argument is not null and that the classes are the same
  - Mix equals does not have to do these checks again

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (super.equals(obj))
  { // the Dog subobjects are equal
    Mix other = (Mix) obj;
    eq = this.breeds.size() == other.breeds.size() &&
         this.breeds.containsAll(other.breeds);
  }
  return eq;
}
```

subclass method that overrides a superclass method can call the original superclass method

Dog toString

```
@Override public String toString()
{
  String s = "size " + this.getSize() +
             "energy " + this.getEnergy();
  return s;
}
```

Mix toString

```
@Override public String toString()
{
  StringBuffer b = new StringBuffer();
  b.append(super.toString());
  for(String s : this.breeds)
    b.append(" " + s);
  b.append(" mix");
  return b.toString();
}
```

size and energy of the dog

breeds of the mix

Dog hashCode

```
// similar to code generated by Eclipse
@Override public int hashCode()
{
  final int prime = 31;
  int result = 1;
  result = prime * result + this.getEnergy();
  result = prime * result + this.getSize();
  return result;
}
```

use this.energy and this.size to compute the hash code

Mix hashCode

```
// similar to code generated by Eclipse
@Override public int hashCode()
{
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + this.breeds.hashCode();
  return result;
}
```

use this.energy, this.size, and this.breeds to compute the hash code

Graphical User Interfaces

notes Chap 7

Java Swing

- Swing is a Java toolkit for building graphical user interfaces (GUIs)
  - http://docs.oracle.com/javase/tutorial/uiswing/TOC.html

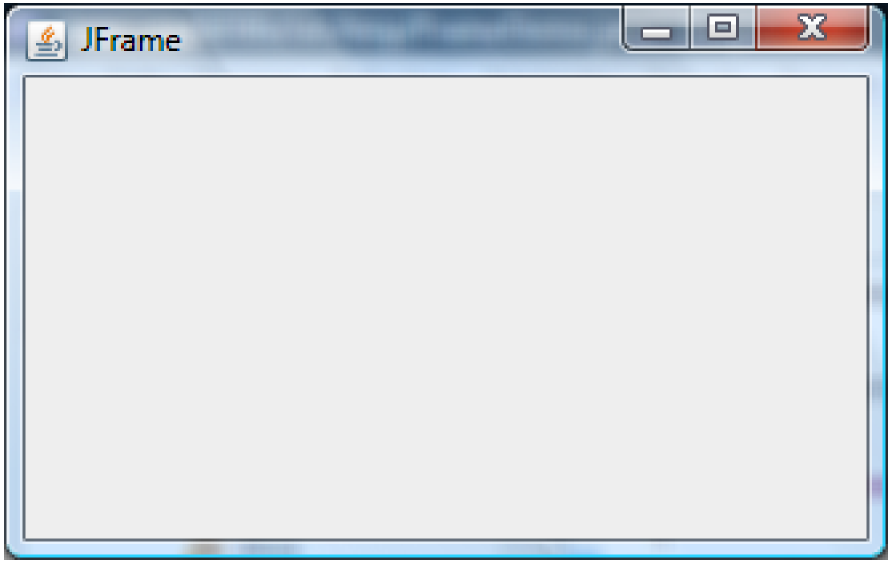
- old version of the Java tutorial had a visual guide of Swing components
  - http://da2i.univ-lille1.fr/doc/tutorial-java/ui/features/components.html

Simple Applications

- simple applications often consist of just a single window (containing some controls)

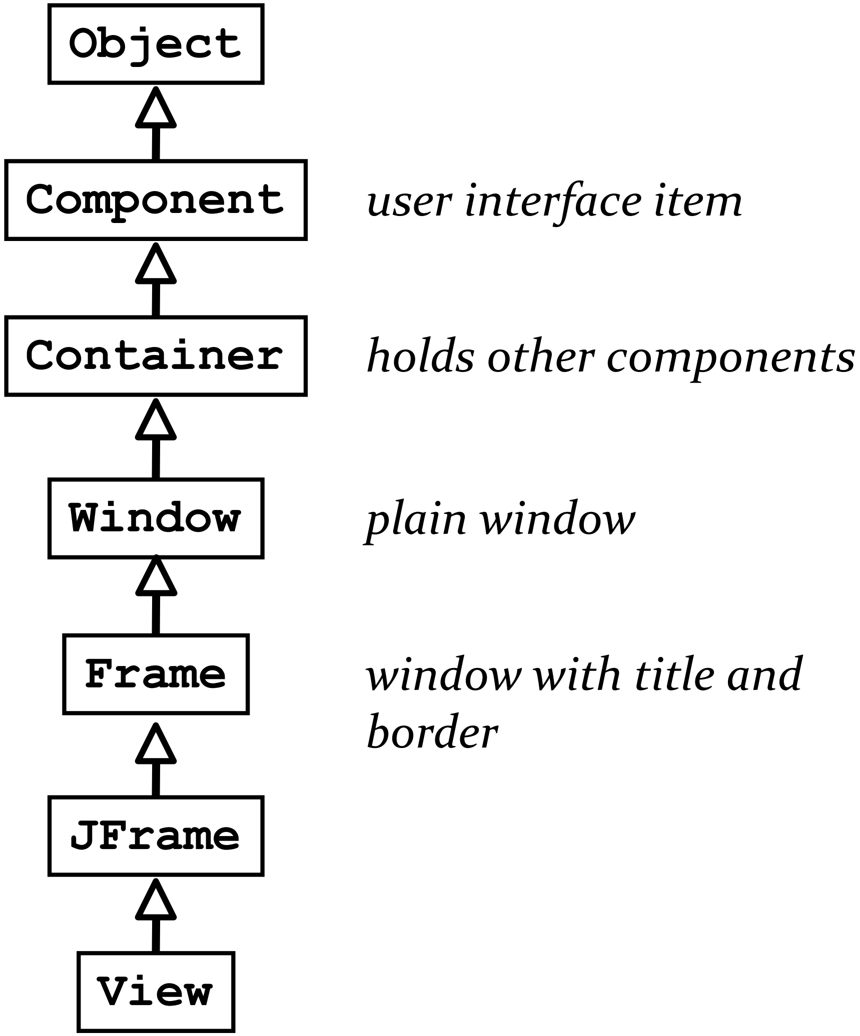
JFrame

window with border, title, buttons



Simple Applications

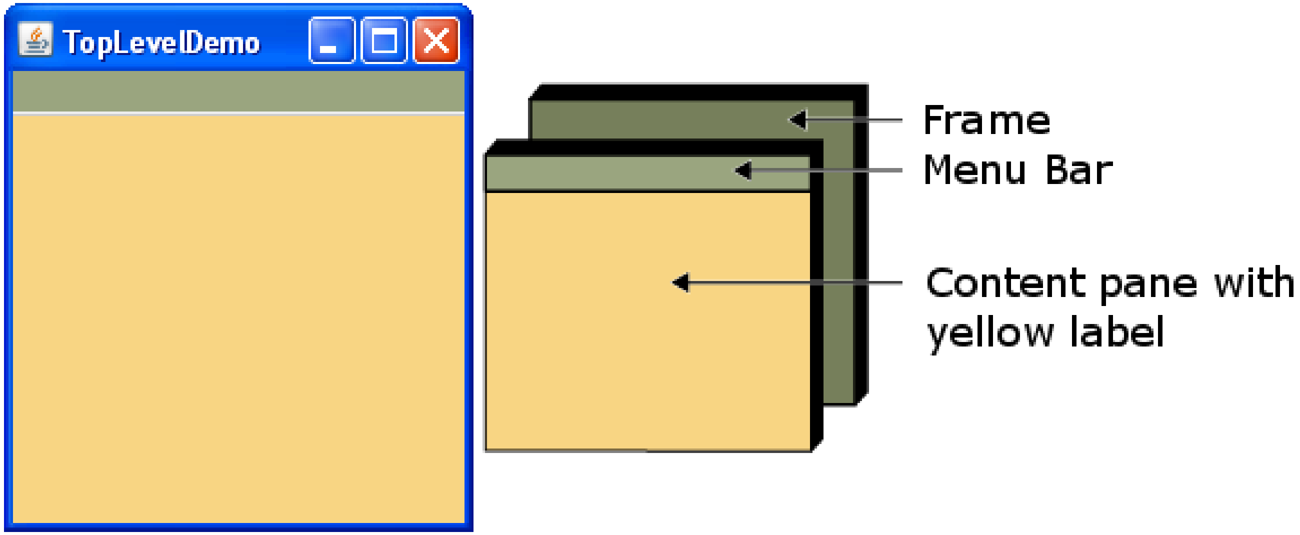
- simple applications can be implemented as a subclass of a JFrame
  - hundreds of inherited methods but only a dozen or so are commonly called by the implementer (see URL below)



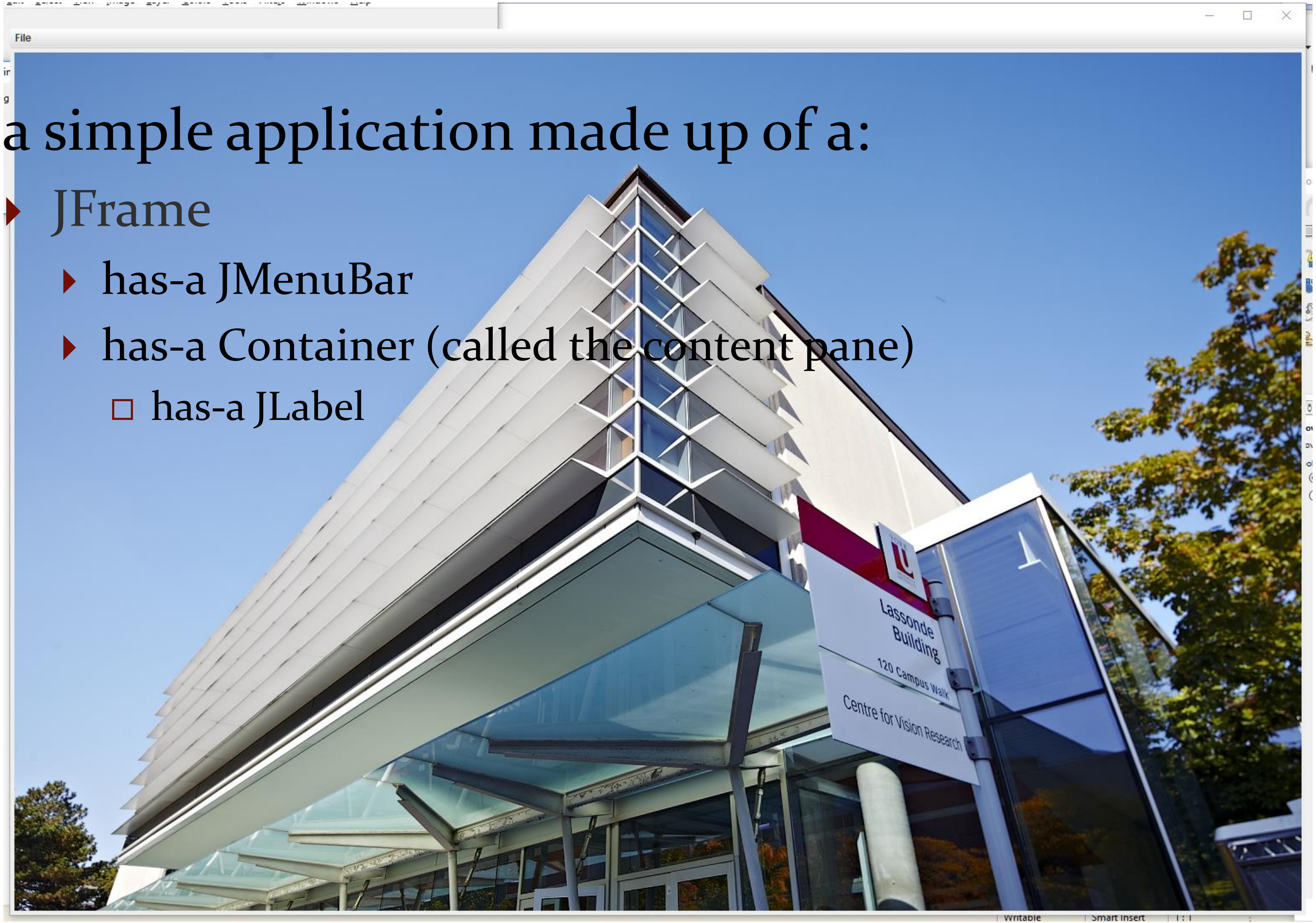
https://docs.oracle.com/javase/tutorial/uiswing/components/frame.html

Simple Applications

- a simple application made up of a:
  - JFrame
    - has-a JMenuBar
    - has-a Container (called the content pane)
      - □ has-a JLabel



Simple Applications



- a simple application made up of a:
  - JFrame
    - has-a JMenuBar
    - has-a Container (called the content pane)
      - □ has-a JLabel

Creating JFrames

1. Create the frame
2. Choose what happens when the frame closes
3. Create components and put them in the frame
4. Size the frame
5. Show it

```
public class ImageViewer extends JFrame                        {

















    }
```

```
public class ImageViewer extends JFrame                            {



    public ImageViewer() {
        // 1. Create the frame
        super("Image Viewer");



    }
```

```
public class ImageViewer extends JFrame                           {



    public ImageViewer() {
        // 1. Create the frame
        super("Image Viewer");

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



    }
```

```
public class ImageViewer extends JFrame implements ActionListener {

    // a unique identifier to associate with the Open command
    public static final String OPEN_COMMAND = "Open";

    // a label to show the image
    private JLabel img;

    public ImageViewer() {
        // 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.makeMenu();
        this.makeLabel();
        this.setLayout(new FlowLayout());




    }
```

to respond to the user selecting the Open command from the menu

controls how the components re-size and re-position when the JFrame changes size

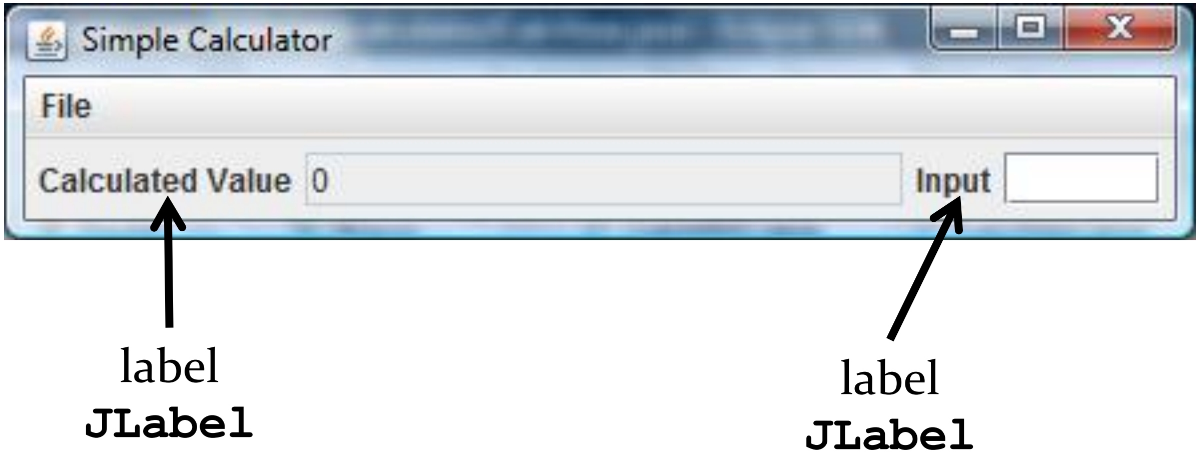
```
public class ImageViewer extends JFrame implements ActionListener {

    // a unique identifier to associate with the Open command
    public static final String OPEN_COMMAND = "Open";

    // a label to show the image
    private JLabel img;

    public ImageViewer() {
        // 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.makeMenu();
        this.makeLabel();
        this.setLayout(new FlowLayout());

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


    }
```

sizes the JFrame so that all components have their preferred size; uses the layout manager to help adjust component sizes

```java
public class ImageViewer extends JFrame implements ActionListener {

    // a unique identifier to associate with the Open command
    public static final String OPEN_COMMAND = "Open";

    // a label to show the image
    private JLabel img;

    public ImageViewer() {
        // 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.makeMenu();
        this.makeLabel();
        this.setLayout(new FlowLayout());

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

        // 5. Show it
        this.setVisible(true);
    }
```

Labels

- a label displays unselectable text and images

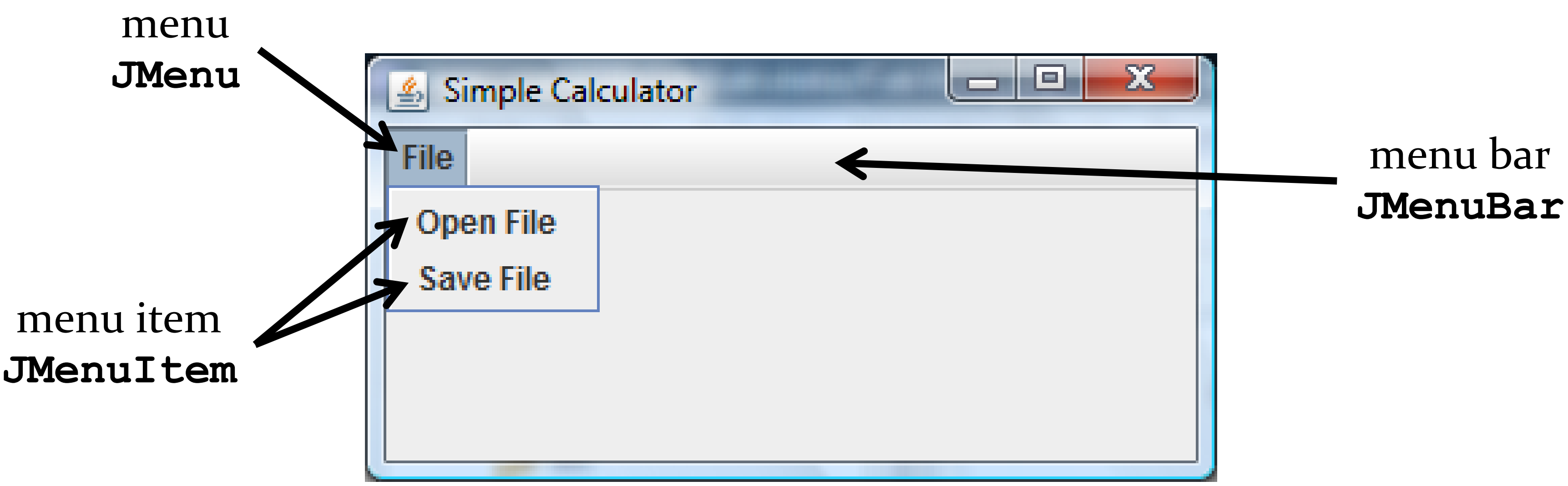


http://docs.oracle.com/javase/tutorial/uiswing/components/label.html

```
private void makeLabel() {
    this.img = new JLabel("");
    this.getContentPane().add(this.img);
}
```

Menus

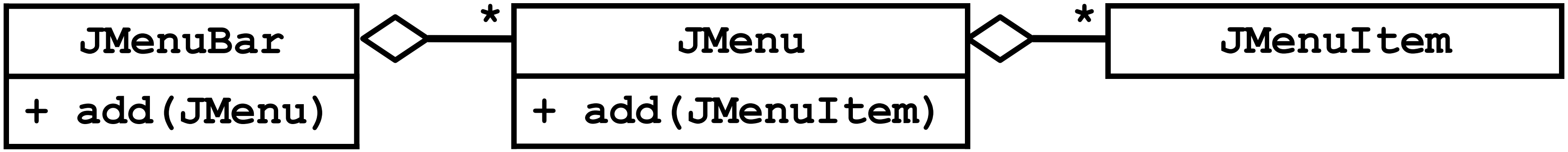
- a menu appears in a *menu bar* (or a popup menu)
- each item in the menu is a *menu item*



menu
`JMenu`

menu bar
`JMenuBar`

menu item
`JMenuItem`



`JMenuBar`
`+ add(JMenu)`

`*`

`JMenu`
`+ add(JMenuItem)`

`*`

`JMenuItem`

http://docs.oracle.com/javase/tutorial/uiswing/components/menu.html

Menus

- to create a menu
  - create a JMenuBar
  - create one or more JMenu objects
    - add the JMenu objects to the JMenuBar
  - create one or more JMenuItem objectes
    - add the JMenuItem objects to the JMenu

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












}
```

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

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





}
```

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

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

    JMenuItem openMenuItem = new JMenuItem("Open...");
    openMenuItem.setActionCommand(ImageViewer.OPEN_COMMAND);
    openMenuItem.addActionListener(this);
    fileMenu.add(openMenuItem);


}
```

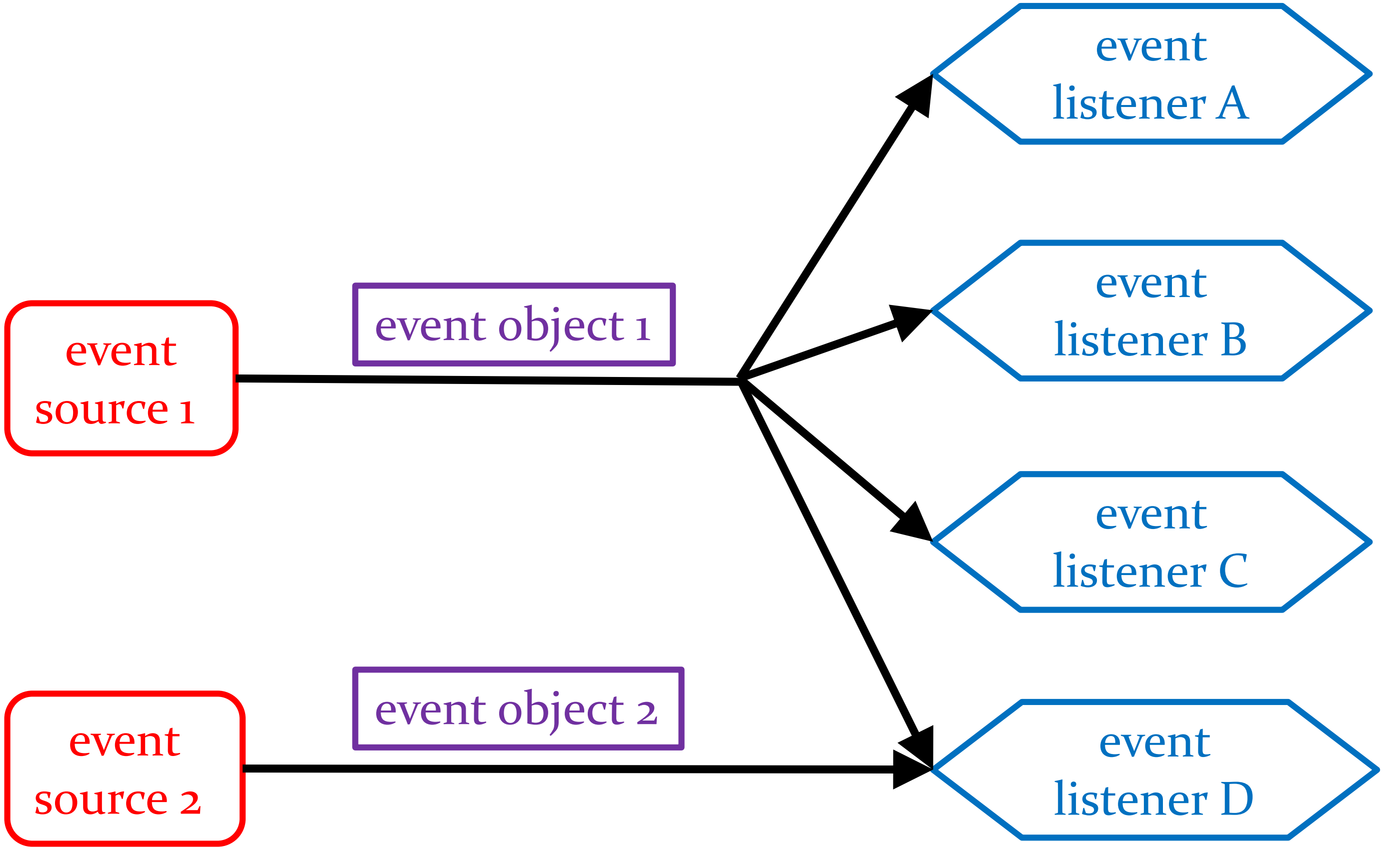
to respond to the user selecting the Open command from the menu

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

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

    JMenuItem openMenuItem = new JMenuItem("Open...");
    openMenuItem.setActionCommand(ImageViewer.OPEN_COMMAND);
    openMenuItem.addActionListener(this);
    fileMenu.add(openMenuItem);

    this.setJMenuBar(menuBar);
}
```

Event Driven Programming

- so far we have a frame with some UI elements (menu, menu item, label)
  - now we need to implement the actions
- each UI element is a source of events
  - button pressed, slider moved, text changed, etc.
- when the user interacts with a UI element an event is triggered
  - this causes an event object to be sent to every object listening for that particular event
    - the event object carries information about the event
- the event listeners respond to the event

Not a UML Diagram



Implementation

- each **`JMenuItem`** has two inherited methods from **`AbstractButton`**

```
public void addActionListener(ActionListener l)

public void setActionCommand(String actionCommand)
```

- for the **`JMenuItem`**

1. call **`addActionListener`** with the listener as the argument
2. call **`setActionCommand`** with a string describing what event has occurred

Implementation

- our application has one event thiat is fired by a button (`JMenuItem`)
  - a button fires an `ActionEvent` event whenever it is clicked
- `ImageViewer` listens for fired `ActionEvents`
  - how? by implementing the `ActionListener` interface

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

```
@Override
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();

}
```

```
@Override
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals(ImageViewer.OPEN_COMMAND)) {



    }
}
```

to respond to the user selecting the Open command from the menu

```
@Override
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals(ImageViewer.OPEN_COMMAND)) {
        JFileChooser chooser = new JFileChooser();










    }
}
```

used to pick the file to open

```
@Override
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals(ImageViewer.OPEN_COMMAND)) {
        JFileChooser chooser = new JFileChooser();
        int result = chooser.showOpenDialog(this);








    }
}
```

show the file chooser and get the user result (ok or cancel)

```
@Override
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals(ImageViewer.OPEN_COMMAND)) {
        JFileChooser chooser = new JFileChooser();
        int result = chooser.showOpenDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {





        }
    }
}
```

user picked a file and pressed ok

```java
@Override
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals(ImageViewer.OPEN_COMMAND)) {
        JFileChooser chooser = new JFileChooser();
        int result = chooser.showOpenDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            String fileName =
                 chooser.getSelectedFile().getAbsolutePath();



        }
    }
}
```

get the file name and directory path that the user picked

```
@Override
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals(ImageViewer.OPEN_COMMAND)) {
        JFileChooser chooser = new JFileChooser();
        int result = chooser.showOpenDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            String fileName =
                 chooser.getSelectedFile().getAbsolutePath();
            ImageIcon icon = new ImageIcon(fileName);




        }
    }
}
```

try to read the image

```
@Override
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals(ImageViewer.OPEN_COMMAND)) {
        JFileChooser chooser = new JFileChooser();
        int result = chooser.showOpenDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            String fileName =
                 chooser.getSelectedFile().getAbsolutePath();
            ImageIcon icon = new ImageIcon(fileName);
            if (icon.getImageLoadStatus() ==
                 MediaTracker.COMPLETE) {


            }
        }
    }
}
```

if the image was
successfully read from disk

```
@Override
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals(ImageViewer.OPEN_COMMAND)) {
        JFileChooser chooser = new JFileChooser();
        int result = chooser.showOpenDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            String fileName =
                 chooser.getSelectedFile().getAbsolutePath();
            ImageIcon icon = new ImageIcon(fileName);
            if (icon.getImageLoadStatus() ==
                 MediaTracker.COMPLETE) {
                this.img.setIcon(icon);
                this.pack();
            }
        }
    }
}
```

set the label image and
re-size the frame

```java
    public static void main(String[] args) {
        // make an ImageViewer instance
        new ImageViewer();
    }
}
```