hi-tech college + JOHN BRYCE Leading in IT Education a matrix company

Advanced Class Features

Abstract Classes, Interfaces and Event-Driven Programming

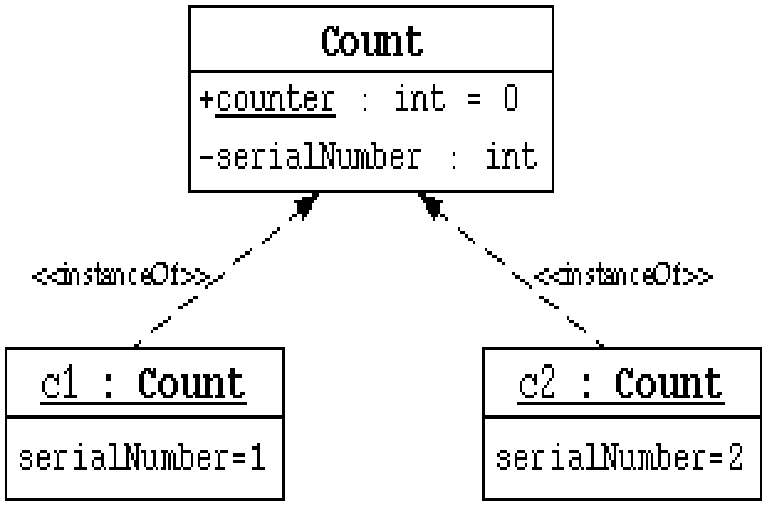
The *static* keyword

- The ***static*** keyword is used as a modifier on variables, methods, and inner classes.
- The ***static*** keyword declares that the attribute or method is associated with the class as a whole rather than any particular instance of that class.
- Thus, static members are often called "class members," such as "class attributes" or "class methods“.

The *static* keyword

- Are shared among all instances of a class

```
1 public class Count {
2   private int serialNumber;
3   public static int counter = 0;
4
5   public Count() {
6     counter++;
7     serialNumber = counter;
8   }
9 }
```



The *static* keyword

- static attribute can be accessed from outside the class if marked as public.

```
1 public class OtherClass {
2   public void incrementNumber() {
3     Count.counter++;          // without an instance of the class
4   }
5 }
```

- You can invoke static method without any instance of the class to which it belongs .
- static method is responsible to access static attributes, and only to them!
- For example, the *main static* method can not access instance variables

The *static* keyword

```
1 public class Count {
2   private int serialNumber;
3   private static int counter = 0;
4
5   public static int getTotalCount()
{
6   return counter;
7   }
8
9   public Count() {
10   counter++;
11   serialNumber = counter;
12  }
13 }
```

```
1  public class TestCounter {
2  public static void main(String[] args) {
3  System.out.println("Number of
   counter is "
4  + Count.getTotalCount());
5  Count count1 = new Count();
6  System.out.println("Number of counter
   is "
7  + Count.getTotalCount());
8   }
9 }
```

The *static* keyword

- A class can contain code in a *static block* that does not exist within a method body.
- Static block code (static initializer, or static constructor) executes only once, when the class is loaded.
- A static block is usually used to initialize static (class) attributes.

```
1 public class Count {
2   public static int counter;
3   static {
4     counter =
   Integer.getInteger("myApp.Count.counter").intValue();
5   }
6 }
```

The *Singleton* Design Pattern

- May be **instantiated only once**
  - The client should not be able to instantiate it:

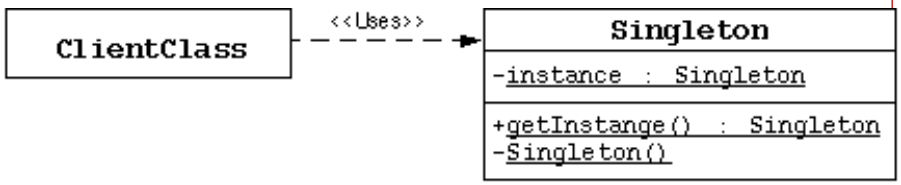
    private constructor
  - The class stores its only instance

    static (class) variable
  - The class lets users to get that only instance:

    static (class) method

Singleton Design Pattern

```
1 public class Company {
2   private static Company instance = new
Company();
3
4   public static Company getCompany() {
5     return instance;
6   }
7
8   private Company() {...}
9
10 }
```



- Client usage:

```
Company c = Company.getCompany();
```

Destructor-like methods

- An object is destructed automatically when it can not be accessed by any reference anymore (its RefCount=0)
- Then, some pre-specified methods are called:
  - finalize – instance method; runs before GC on the instance

The *static* and *final* Keywords Combination

- Constants in Java are specified as *final*
- Usually, they are are also static
  - Math.PI
  - Math.E
- Globally accessible mathematical functions are also static:
  - Math.sin()
  - Math.cos()

The *final* keyword

- Constants:

  private static **final** double DEFAULT_INTEREST_RATE=3.2;

- Blank Final Instance Attribute:

```
public class Customer {
  private final long customerID;
  public Customer() {
    customerID = createID();
  }
  public long getID() {
    return customerID;
  }
  private long createID() {
    return ... // generate new ID
  }
... // more declarations }
```

The *final* Keyword Summary

- You can not subclass a *final* class
- You can not override a *final* method
- A *final* variable is a constant
- You can set a *final* variable only once, but that assignment can occur independently of the declaration; this is called "blank final variable"
  - A blank *final* instance attribute must be set in every constructor
  - A blank final method variable must be set in the method body before being used

Abstract classes

- A class that can not be instantiated
- A class that declared as abstract:
  `public abstract class Shape {...`
- We declare a class as *abstract* because we:
  - want to prohibit it from being instantiated
  - lack the functionality to implement some methods
- Can not be both *final* and *abstract* at the same time

Abstract class example

```
public abstract class Shape {
    public abstract void paint();//no body
}
public class Circle extends Shape {
    public void paint(){ //implement the
                         //body
        ... draw a circle
    }
}
```

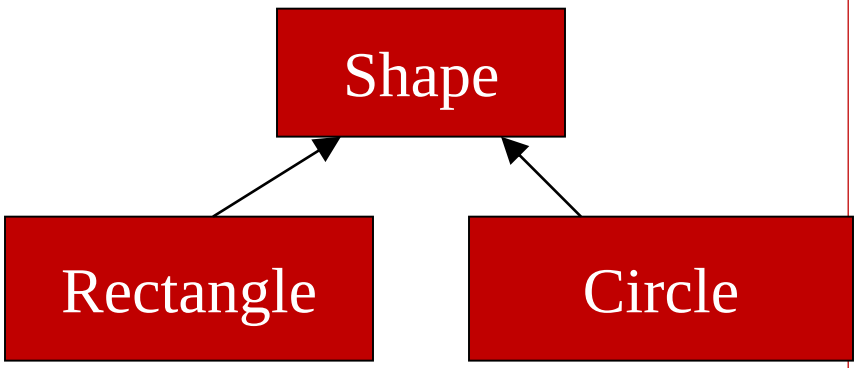
- For an undefined `Shape` we don't know how to paint it!

Abstract methods

- Abstract classes may (but don't have to) have abstract methods
- These methods must be overridden in non-abstract subclasses to provide an implementation
- A method can not be both *final* and *abstract* at the same time

Abstract Classes Usage Recommendations

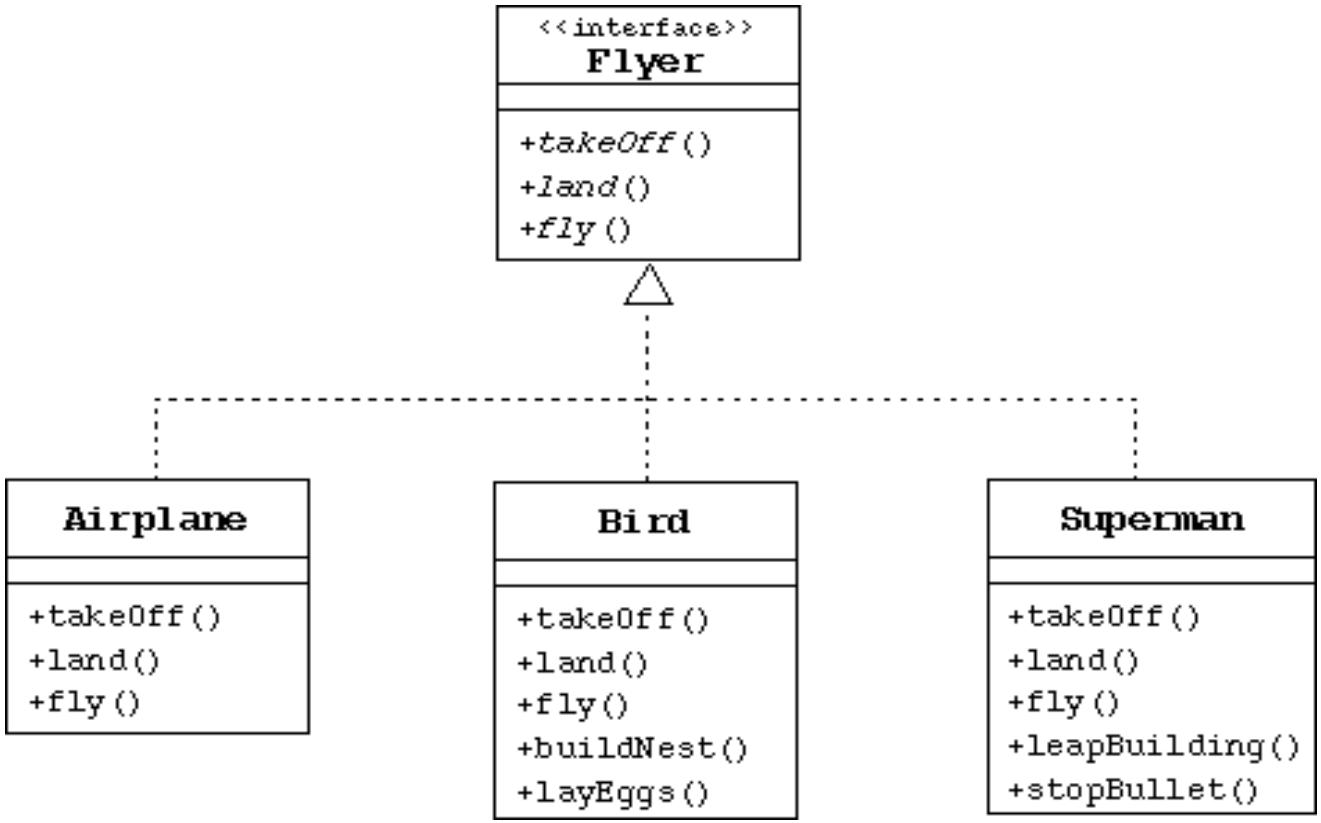
- Instead of mixing different levels of inheritance:
  - **Use only Rectangle and Circle** (both – 2nd depth)
  - **Do not use Shape** directly
- Importance in superclassing for heterogeneous collections
  - Define a list of different shapes, with the same methods, may be implemented differently
  - getPerimeter() is undefined at the higher level



Interfaces

- A "public interface" is a contract between client code and the class that implements that interface
- A Java *interface* is a formal declaration of such contract in which all methods contain no implementation
- Many, unrelated classes can implement the same interface
- A class can implement many, unrelated interfaces

Interfaces


<<interface>>
Flyer
+takeOff()
+land()
+fly()
Airplane
+takeOff()
+land()
+fly()
Bird
+takeOff()
+land()
+fly()
+buildNest()
+layEggs()
Superman
+takeOff()
+land()
+fly()
+leapBuilding()
+stopBullet()

Interfaces

- An interface resembles to an abstract class but it is not a class
- An interface can not have runnable code inside
- All interfaces and their members are public and abstract by default, thus, it is deprecated to state these explicitly

Interface members

- Interfaces can contain:
  - final variables (constants)
  - abstract methods
- A non-abstract class implements an interface if and only if:
  - it declares that it implements the interface
  - it implements all of the abstract methods of the interface

Interface references

- An interface type variable can be assigned instances of classes implementing the interface
- An abstract class can implement an interface. Hence, it declares some or all of the interface's methods as abstract.

Considerable example

```
public interface Paintable {
    public void paint();
}
public abstract class Shape implements Paintable {
    public abstract void paint(); // no body
}
...
Shape s = new Circle(5.0);
Paintable p = s;
p = new Circle(5.0);//Circle implements Paintable
```

Interfaces and inheritance

- A subclass implements all the interfaces that its superclass implements
- A class can implement more than one interface
- An interface can extend multiple interfaces
- A class implementing an interface A implements all the interfaces that A extends

Interface inheritance

```
public interface CanSayYes {
    public void sayYes();
}
public interface CanSayNo {
    public void sayNo();
}
public interface CanSayYesOrNo extends CanSayYes, CanSayNo{
}
public class Politician implements CanSayYesOrNo {
    public void sayYes() {System.out.println(”yes”);}
    public void sayNo()  {System.out.println(”no”);}
}
```

Uses Of Interfaces

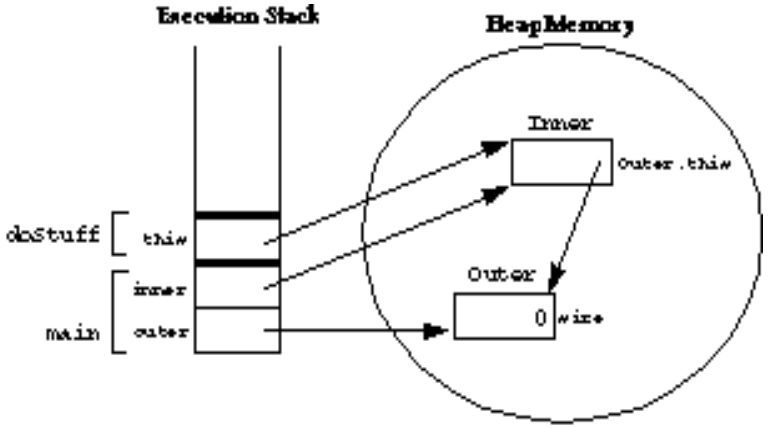
- Declaring methods which one or more classes are expected to implement
- Determining an object's programming interface without revealing the actual body of the class
- Capturing **similarities** between unrelated classes without forcing a class relationship (has a relationship)
- Simulating **multiple inheritance** by declaring a class that implements several interfaces

Inner classes

- Allow a class definition to be placed inside another class definition
- Group classes that logically belong together
- Have an access to their enclosing class’s scope

Inner Classes

```
1   public class Outer2 {
2     private int size;
3
4     public class Inner {
5       public void doStuff() {
6         size++;
7       }
8     }
9   }
```




```
1   public class TestInner {
2     public static void main(String[] args) {
3       Outer2 outer = new Outer2();
4
5       // Must create an Inner object relative to an Outer
6       Outer2.Inner inner = outer.new Inner();
7       inner.doStuff();
8     }
9   }
```

Inner Classes

- You can use the class name only within the defined scope, except when used in a qualified name. The name of the inner class must differ from the enclosing class.
- The inner class can be defined inside a method. Only local variables marked as final can be accessed by methods within an inner class.

Inner Classes

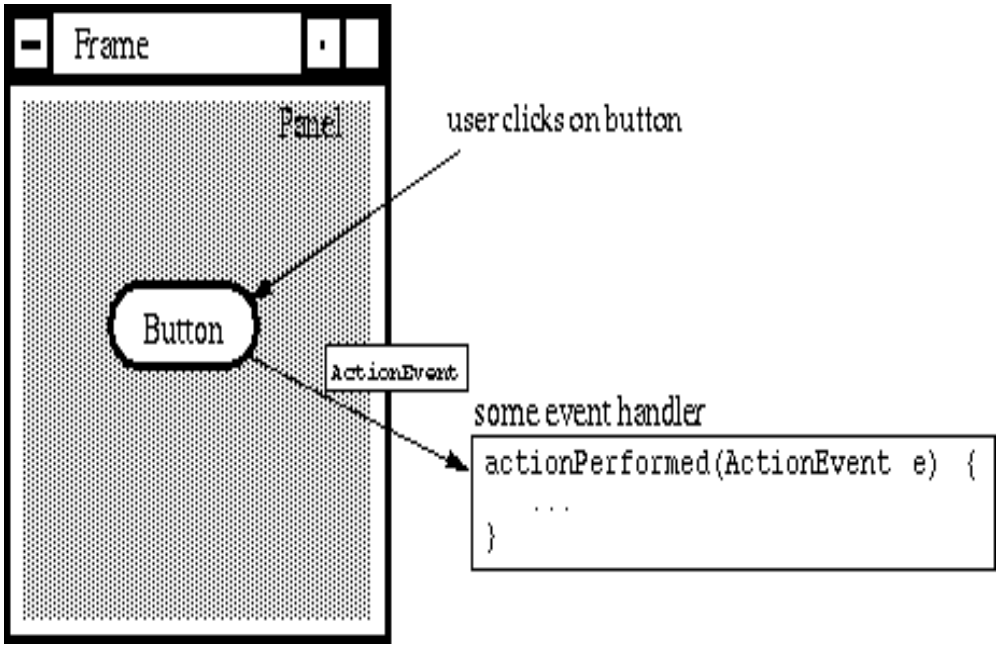
- The inner class can use both class and instance variables of the enclosing classes and local variables of enclosing blocks
- The inner class can be defined as abstract
- The inner class can have any access mode
- The inner class can act as an interface implemented by another inner class

Inner Classes

- Inner classes that are declared as static automatically become top-level classes
- Inner classes can not declare any static members; only top-level classes can declare static members
- An inner class wanting to use a static member must be declared static

Event-driven Programming and *Listeners*

- Events - Objects that describe what happened
- Event sources - The generator of an event
- Event handlers - A method that receives an event object, deciphers it, and processes the user's interaction
- Event handlers register with components when they are interested in events generated by that component



Event-driven Programming and Listeners

```
public class TestButton {
  private Button b;

  public void launchFrame() {
   b.addActionListener(new ButtonHandler());
  }

  public class ButtonHandler implements
ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Action occurred");
      System.out.println("Button's command is: "
        + e.getActionCommand());
    }
  } //inner class

} //outer class
```

```
public class TestButton {
private Button b;

public void launchFrame() {
 b.addActionListener(new ActionListener()
   {
     public void actionPerformed(ActionEvent e) {
       System.out.println("Action occurred");
       System.out.println("Button's command is: "
         + e.getActionCommand());
     }
  } //anonymous inner class
);
}

 } //outer class
```

Event-driven Programming and Listeners

- **Listener** – interface declaring the methods to be called when some events occur
  - After a specific class implementing this interface, is registered, its methods will be called
  - b.addActionListener(…)
- **Event adapter** – specific (non-abstract) classes implementing the listener interfaces in an empty way
  - No need to implement ALL the interface methods
  - WindowListener has 7 methods, when you may be interested in listening to only one event: windowClosing
  - Without an adapter, your class could be abstract!

References

- http://java.sun.com/javase/6/docs/technotes/guides
- SUN Educational Services SL-275
  download:
  http://www.sun.com/products-n-solutions/edu/programs/sai/download/SL275.E.2.desc.pdf