ILLINOIS INSTITUTE
OF TECHNOLOGY

Transforming Lives.Inventing the Future.**www.iit.edu**

- Abstract Classes and Methods
- Interfaces

abstract Classes and Methods

- An **abstract class** is a class that is not completely implemented.
- Usually, the abstract class contains at least one **abstract method.**
  - An abstract method specifies an API but <u>does not provide an implementation</u>.
  - The abstract method is used as a pattern for a method the subclasses should implement.

More on *abstract* Classes

- An object reference to an *abstract* class can be declared.
  - We use this capability in polymorphism, discussed later.
- An *abstract* class cannot be used to instantiate objects (because the class is not complete).
- An *abstract* class can be extended.
  - subclasses can complete the implementation and objects of those subclasses can be instantiated.

Defining an *abstract* class

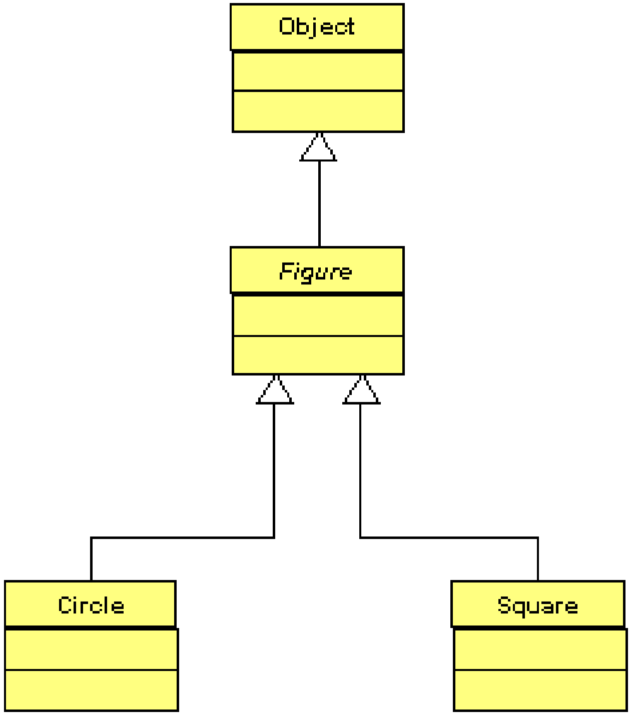
- To declare a class as *abstract*, include the *abstract* keyword in the class header:

```
accessModifier abstract class ClassName
{
   // class body
}
```

```
public abstract class Figure
 {
   private int x;
   private int y;
   private Color color;
   …
   // abstract draw method
   public abstract void draw( Graphics g );
 }
```

Example Hierarchy

- We can define a Figure hierarchy.
- The superclass is *Figure*, which is *abstract.*
  - In the UML diagram, *Figure* is set in italics to indicate that it is *abstract*.
- We will derive two concrete subclasses:
  - Circle and
  - Square.



The *Figure* Class

```
public abstract class Figure
{
 protected int x;
 protected int y;
 protected Color color;
   …
   // abstract draw method
   public abstract void draw( Graphics g );
}
```

- All classes in the hierarchy will inherit an (x, y) coordinate and color.
- Subclasses MUST implement the *draw* method.

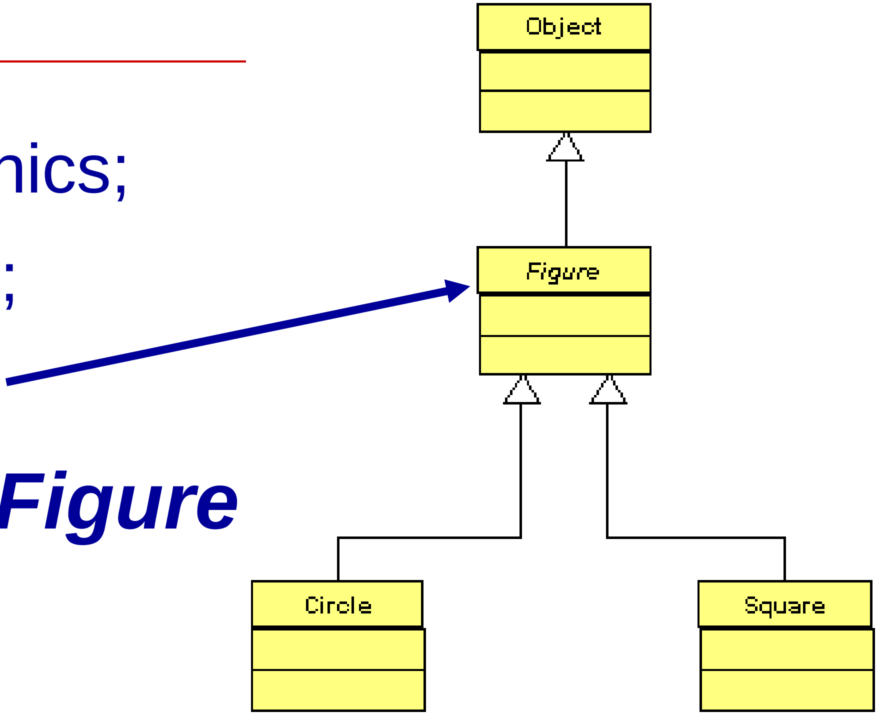
Subclasses of *abstract* Classes

- A subclass of an abstract class can implement all, some, or none of the *abstract* methods.

- If the subclass does not implement all of the *abstract* methods, it must also be declared as *abstract*.
  - Our *Circle* subclass adds a *radius* instance variable and implements the ***draw*** method.
  - Our *Square* subclass adds a *length* instance variable and implements the ***draw*** method.

- *See Examples Figure.java,*

Figure.java 1/3



```
import java.awt.Graphics;
import java.awt.Color;


public abstract class Figure
        private int x, y;
        private Color color;
        /** default constructor */
        public Figure( )   {
          x = 0;      y = 0;      color = Color.BLACK;
        }
```

Figure.java cont. 2/3

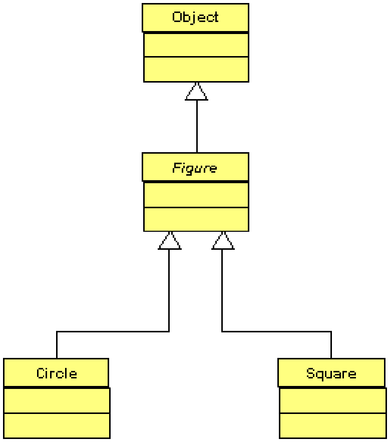
```
 /** overloaded constructor */
public Figure( int startX, int startY, Color startColor )
{   x = startX;   y = startY;   color = startColor; }

public Color getColor( )
{  Color tempColor = color;return tempColor; }

public void setColor( Color newColor )
{  color = newColor;   }
```

Figure.java cont. 3/3

```
  public int getX( )   {   return x;  }

   public void setX( int newX )   {  x = newX; }

   public int getY( )  { return y;  }

   public void setY( int newY )  {  y = newY;  }

public abstract void draw( Graphics g );
}
```

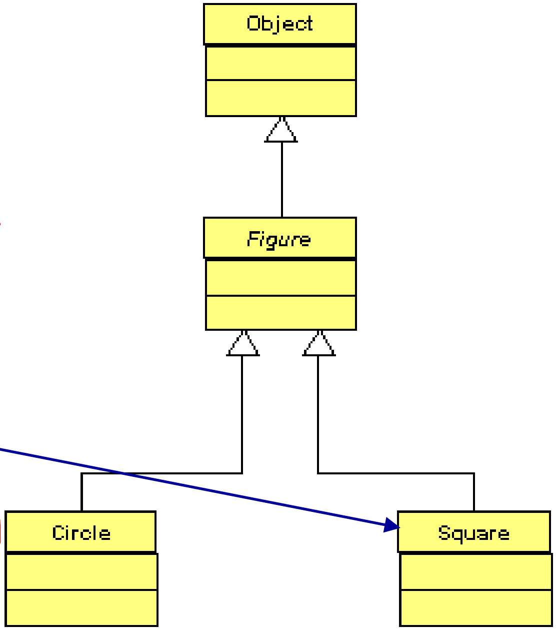
Circle.java

- public class **Circle extends *Figure*** {
- private int **diameter**;
- public **Circle**() { x = 0; y = 0; diameter = 10; color = C
- public **Circle**(int sX, int sY, int sDiameter, Color sColor)
- { x = sX; y = sY; diameter = sDiameter; color = sColor; }
- public int **getDiameter**( ) { return diameter; }
- public void **setDiameter**(int newDiameter) { diameter = newDiameter; }
- **public void draw(Graphics g) {**
- **g.setColor(color);**
- **g.fillOval(x, y, diameter, diameter);**
- **}**
- }



Square.java

```
public class Square extends Figure {
  private int length;
  public Square( ) {    super( );  length = 0;    }
  public Square( int startX, int startY, Color startColo
     super( startX, startY, startColor );
     setLength( startLength );
  }
  public void setLength( int newLength ) { length = newLength; }
  public int getLength( ) { return length; }
  public void draw( Graphics g ) {
       g.setColor( getColor( ) );
       g.fillRect( getX( ), getY( ),
                   length, length );
   }
}
```



TrafficLight.java No Polymorphism

```
public class TrafficLight extends JApplet {
private ArrayList<Circle> circlesList;
private ArrayList<Square> squaresList;
public void init( )   {
squaresList = new ArrayList<Square>( );
squaresList.add( new Square( 150, 100, Color.BLACK, 40 ) );
squaresList.add( new Square( 150, 140, Color.BLACK, 40 ) );
squaresList.add( new Square( 150, 180, Color.BLACK, 40 ) );
circlesList = new ArrayList<Circle>( );
circlesList.add( new Circle( 160, 110, Color.RED, 10 ) );
circlesList.add( new Circle( 160, 150, Color.YELLOW, 10 ) );
circlesList.add( new Circle( 160, 190, Color.GREEN, 10 ) );
 }
public void paint( Graphics g ) {
   for ( Square s : squaresList )
     s.draw( g );

   for ( Circle c : circlesList )
     c.draw( g );
 }
}
```

Restrictions for Defining - *abstract* Classes

- Classes must be declared *abstract* if the class contains any *abstract* methods.
- *abstract* classes can be extended.
- An object reference to an *abstract* class can be declared.
- *abstract* classes cannot be used to instantiate objects.
- *abstract* methods can be declared only within an *abstract* class.
- An *abstract* method must consist of a method header followed by a semicolon.
- *abstract* methods cannot be called.
- *abstract* methods cannot be declared as *private* or *static.*
- A constructor cannot be declared *abstract.*

Final Methods and Classes

- A method that is declared final can’t be overridden

- A class that is declared final can’t be a superclass
  - All methods in a a final class are final

Topics

- Polymorphism
- Interfaces

Polymorphism

- An important concept in inheritance is that an object of a subclass is also an object of any of its superclasses.

- That concept is the basis for an important OOP feature, called **polymorphism**.

- Polymorphism simplifies the processing of various objects in the same class hierarchy because we can use the same method call for any object in the hierarchy using a superclass object reference.

Polymorphism Requirements

- To use polymorphism, these conditions must be true:
  - the classes are in the same hierarchy.
  - all subclasses override the same method.
  - a subclass object reference is assigned to a superclass object reference.
  - the superclass object reference is used to call the method.

Example

- Example **TrafficLightPolymorphism.java** shows how we can simplify the drawing of *Circle* and *Square* objects.
  - We instantiate a *Figure ArrayList* and add *Circle* and *Square* objects to it.

```
ArrayList<Figure> figuresList
           = new ArrayList<Figure>( );
figuresList.add( new Square( 150, 100,
                  Color.BLACK, 40 ) );
figuresList.add( new Circle( 160, 110,
                   Color.RED, 10 ) );
```

…

- In the *paint* method, we call *draw* this way:

```
for ( Figure f : figuresList )
        f.draw( g );
```

Polymorphism Conditions

- Example **TrafficLightPolymorphism.java** shows that we have fulfilled the conditions for polymorphism:
  - The *Figure*, *Circle,* and *Square* classes are in the same hierarchy.
  - The non-abstract **Circle** and **Square** classes implement the *draw* method.
  - We assigned the **Circle** and **Square** objects to *Figure* references.
  - We called the *draw* method using *Figure* references.

TrafficLightPolymorphism.java

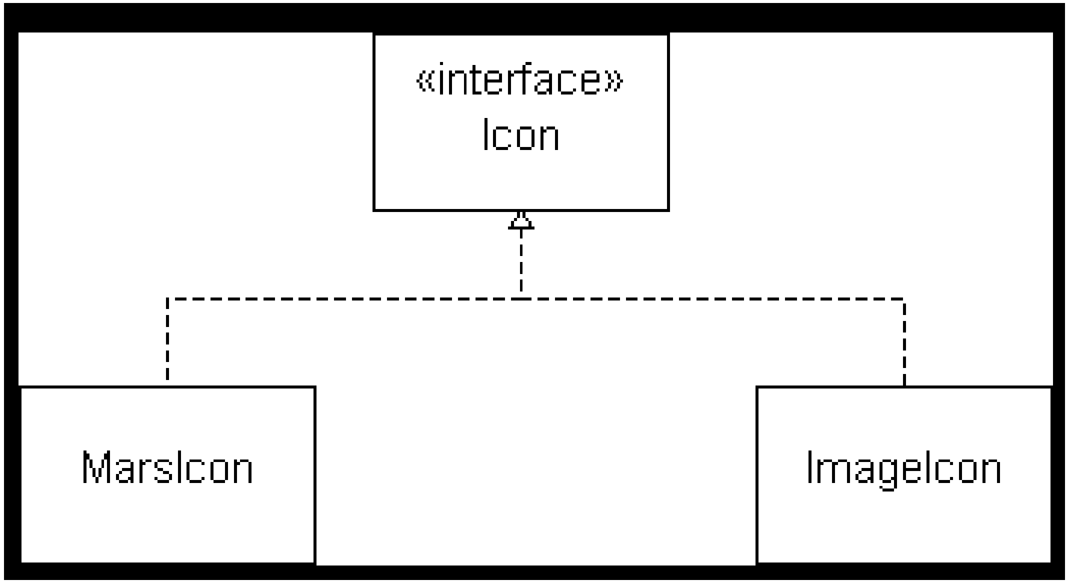
```
public class TrafficLightPolymorphism extends JApplet {
  private ArrayList<Figure> figuresList;

  public void init( )   {
    figuresList = new ArrayList<Figure>( );
    figuresList.add( new Square( 150, 100, Color.BLACK, 40 ) );
    figuresList.add( new Circle( 160, 110, Color.RED, 10 ) );

    figuresList.add( new Square( 150, 140, Color.BLACK, 40 ) );
    figuresList.add( new Circle( 160, 150, Color.YELLOW, 10 ) );

    figuresList.add( new Square( 150, 180, Color.BLACK, 40 ) );
    figuresList.add( new Circle( 160, 190, Color.GREEN, 10 ) );
  }
  public void paint( Graphics g )   {
    for ( Figure f : figuresList )
         f.draw( g );
  }
}
```

Interfaces

- Interfaces

Modeling an Interface

- An interface and a class that implements the interface model the “*is-a*” relationship
- In the following UML diagram, MarsIcon is an Icon, and ImageIcon is an Icon.



Interfaces

- A class can inherit directly from only one class, that is, a class can *extend* only one class.
- To allow a class to inherit behavior from multiple sources, Java provides the **interface**.
- An interface is a group of related methods with empty bodies.
  - A named set of operations
- An interface typically specifies behavior that a class will *implement*.
- Interface members can be any of the following:

  classes, **constants**, ***abstract* methods** and other interfaces
- Interface members can NOT be instance variables.
- All methods in an interface are abstract.

Interface Syntax

- To define an interface, use the following syntax:

```
accessModifier interface InterfaceName
{     // body of interface }
```

- All interfaces are *abstract*; thus, they cannot be instantiated. The *abstract* keyword, however, can be omitted in the interface definition.
- If the interface access modifier are public, all its methods are public as well
- An interface's fields are *public*, *static*, and *final.* These keywords can be specified or omitted.
- When you define a field in an interface, you must assign a value to the field.
- All methods within an interface must be *abstract.* The ***abstract*** keyword also can be omitted from the method definition.

Inheriting from an Interface

- To inherit from an interface, a class declares that it **implements** the interface in the class definition, using the following syntax:

```
accessModifier class ClassName
    extends SuperclassName
     implements Interface1, Interface2, …
```

- The *extends* clause is optional.
- A class can *implement* 0, 1, or more interfaces.

Inheriting from an Interface

- A class can *implement* 0, 1, or more interfaces.
  - When a class *implements* an interface, the class **<u>must</u>** provide an implementation **for each** method in the interface.

- Implementing an interface allows a class to become more formal about the behavior it promises to provide.

- Interfaces form a <u>contract</u> between the class and the outside world, and this contract is enforced at build time by the compiler.
  - If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

Example

- **Define an *abstract* class *Animal* with one *abstract* method (See Example Animal.java):**

```
 public abstract class Animal {
     private int x;      private int y;      private String ID;
      public Animal( )  { ID = ""; }
      public Animal( String rID, int rX, int rY )
                  { ID = rID; x = rX; y = rY;  }
      public String getID( ) { return ID; }
      public int getX( ) { return x; }
      public int getY( ) { return y; }
      public void setX( int newX ) { x = newX; }
      public void setY( int newY ) { y = newY; }
       public abstract void draw( Graphics g );
```

- }

Example

- Define a *Moveable* interface with one abstract method:

```
public interface Moveable
   {
      int FAST = 5; // static constant
      int SLOW = 1; // static constant

      void move( ); // abstract method
   }
```

Derived Classes

- *TortoiseRacer* class
  - *extends Animal* class
  - *implements Moveable* interface
  - implements *draw* and *move* methods
- *TortoiseNonRacer* class
  - *extends Animal* class
  - (does not implement *Moveable* interface)
  - implements *draw* method only
- *See Examples Animal.java, Moveable.java, TortoiseRacer , & TortoiseRacerClient.java*

TortoiseRacer.java

```
public class TortoiseRacer extends Animal implements Moveable  {
  public TortoiseRacer( ) { super( ); }
  public TortoiseRacer( String rID, int rX, int rY ) {
    super( rID, rX, rY );
  }
  public void draw( Graphics g ) {
    int startX = getX( );    int startY = getY( );
    g.setColor( new Color( 34, 139, 34 ) );
   g.fillOval( startX, startY, 25, 15 );
    g.fillOval( startX + 20, startY + 5,  15, 10 );
     g.clearRect( startX, startY + 11, 35, 4 );
    //feet
    g.setColor( new Color( 34, 139, 34 ) );  // brown
    g.fillOval( startX + 3, startY + 10,  5, 5 );
    g.fillOval( startX + 17, startY + 10, 5, 5 );
  }
   public void move( ) { setX( getX( ) + SLOW ); }
}
```

TortoiseRacerClient.java

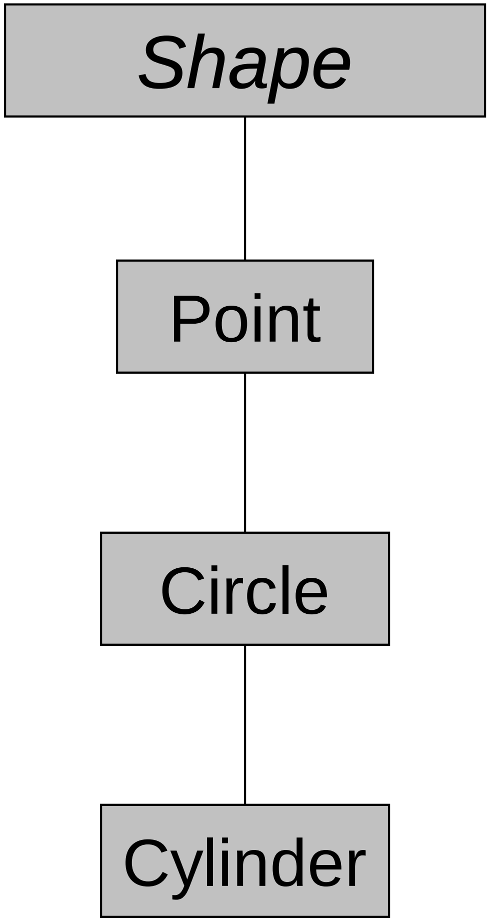
```
public class TortoiseRacerClient extends JApplet {
  private TortoiseRacer t;

  public void init( ) {
    t = new TortoiseRacer( "Tortoise", 50, 50 );
  }
  public void paint( Graphics g ) {
    for ( int i = 0; i < getWidth( ); i++ )      {
      t.move( );
      t.draw( g );

     Pause.wait( .03 );
     g.clearRect( 0, 0, getWidth( ), getHeight( ) );
    }
  }
}
```

Shape Example

- Students Only

Abstract Classes Example

- Shape
  - Defines all methods that are common to our shapes
- Point
  - Inherits these methods
- Circle
  - Inherits some and overrides some other methods
- Cylinder
  - Inherits some and overrides some other methods

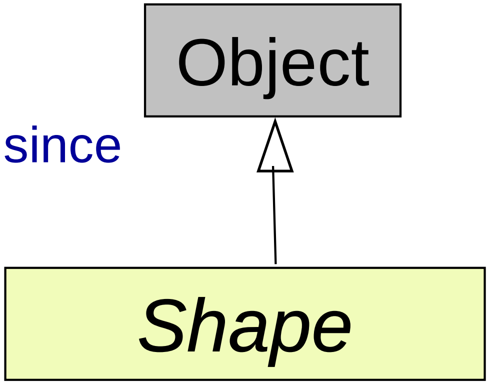


Shape

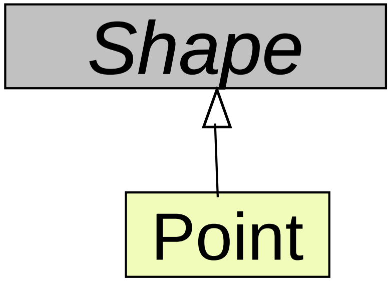
- Shape is an abstract superclass
- It still contain implementations of methods area and volume which are inheritable
  - Shape provide an inheritable interface (set of services)
  - All subclasses can use or override these interfaces (methods)
- The point here is that subclasses can inherit interface and/or implementation from a supperclass

Shape Example: Shape Class

```
public abstract class Shape extends Object {

   // return shape's area , overridden when it make since
   public double area()    {
      return 0.0;
   }
   // return shape's volume, overridden when it make since
   public double volume()    {
      return 0.0;
   }
   // abstract method must be overridden by all concrete
   // subclasses to return appropriate shape name
   public abstract String getName();
}  // end class Shape
```



Shape Example: Point Class 1/2



```
public class Point extends Shape {
   protected int x, y; // coordinates of the Point

public Point()     {   setPoint( 0, 0 );     }

public Point( int xCoordinate, int yCoordinate )
    { setPoint( xCoordinate, yCoordinate ); }

public void setPoint( int xCoordinate, int yCoordinate )
   { x = xCoordinate;  y = yCoordinate;  }

public int getX()    {  return x;  }
```

Point inherits (NOT override) both volume and area methods of shape (zero)

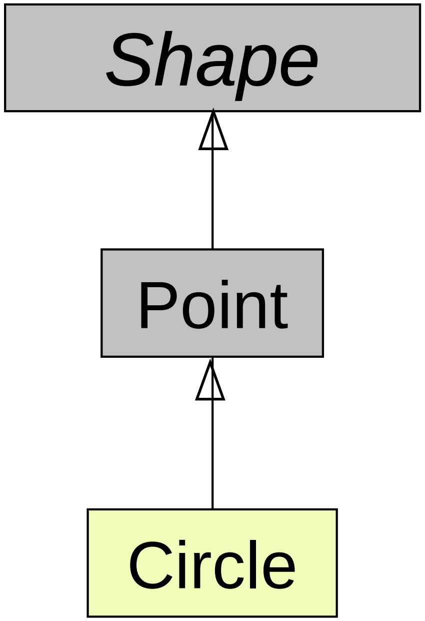
Shape Example: Point Class 2/2

```
public int getY()    {
     return y;
  }
     // convert point into String representation
public String toString()    {
          return "[" + x + ", " + y + "]";
  }
    // return shape name, an implementation of the abstract method
public String getName()    {
          return "Point";
       }
    }  // end class Point
```

If getName is not defined here, then point would have been an abstract class and no objects of it can be instantiated

Shape Example: Circle Class 1/2

```
public class Circle extends Point {  // inherits from Point
  protected double radius;
public Circle()    {
    // implicit call to superclass constructor here
    setRadius( 0 );
  }
public Circle( double circleRadius, int xCoordinate,  int
   yCoordinate )  {
    // call superclass constructor
    super( xCoordinate, yCoordinate );
    setRadius( circleRadius );
  }
public void setRadius( double circleRadius )
   { radius = ( circleRadius >= 0 ? circleRadius : 0 );  }

public double getRadius()   { return radius; }
```



Circle inherits the volume method from point(zero) and overrides the area method

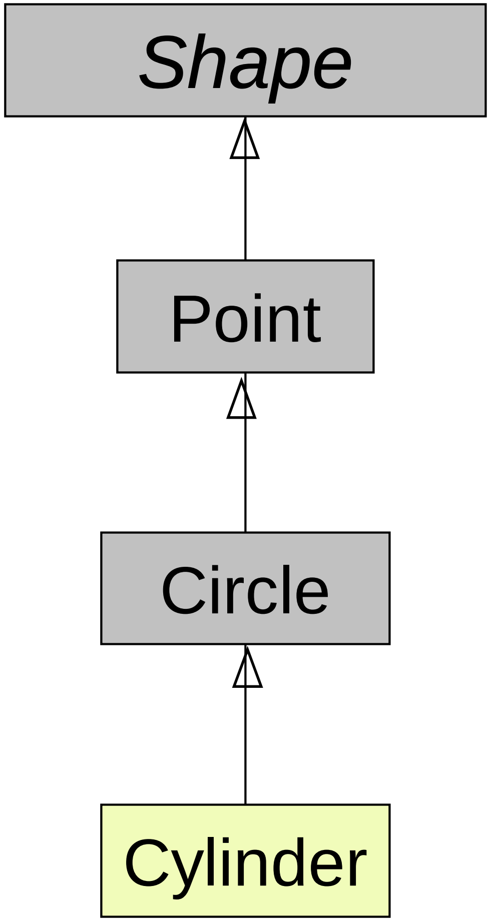
Shape Example: Circle Class 2/2

```
// calculate area of Circle, overrides area of Shape
   public double area()    {
      return Math.PI * radius * radius;
   }

   // convert Circle to a String represention
   public String toString()    {
      return "Center = " + super.toString() +
         "; Radius = " + radius;
   }
public String getName(
      return "Circle";
   }
}  // end class Circle
```

If getName is not defined here, then area() version of Point class would be inherited

Shape Example: Cylinder Class 1/2

```
public class Cylinder extends Circle {
   protected double height;  // height of Cylinder
public Cylinder()    {
  setHeight( 0 );
}
public Cylinder( double cylinderHeight, double
    cylinderRadius, int xCoordinate,   int yCoordinate
    )    {
      super( cylinderRadius, xCoordinate,
    yCoordinate );
      setHeight( cylinderHeight );
}
public void setHeight( double cylinderHeight )   {
      height = ( cylinderHeight >= 0 ? cylinderHeight :
    0 );
}
public double getHeight()  {
      return height;
   }
```



Cylinder overrides both volume and area methods

Shape Example: Cylinder Class 2/2

```
public double area()   {
        return 2 * super.area() + 2 * Math.PI * radius * height;
    }
public double volume()     {
        return super.area() * height;
    }
public String toString()    {
        return super.toString() + "; Height = " + height;
    }
public String getName()    {
        return "Cylinder";
    }
}  // end class Cylinder
```

If getName is not defined here, then area() version of Circle class would be inherited

Shape Example: Test Class 1/3

```
import javax.swing.JOptionPane;
public class Test {              // test Shape hierarchy
   public static void main( String args[] )
   {   // create shapes
      Point point = new Point( 7, 11 );
      Circle circle = new Circle( 3.5, 22, 8 );
      Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );
      // create Shape array
      Shape arrayOfShapes[] = new Shape[ 3 ];
      // aim arrayOfShapes[ 0 ] at subclass Point object
      arrayOfShapes[ 0 ] = point;
      // aim arrayOfShapes[ 1 ] at subclass Circle object
      arrayOfShapes[ 1 ] = circle;
      // aim arrayOfShapes[ 2 ] at subclass Cylinder object
      arrayOfShapes[ 2 ] = cylinder;
```

Shape Example: Test Class 2/3

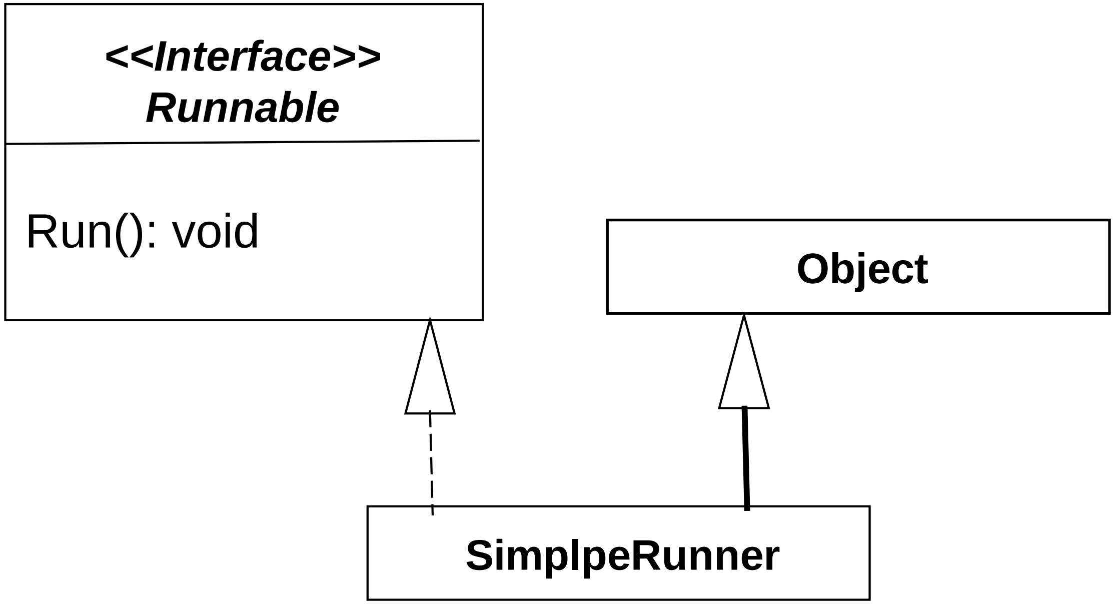
```
// get name and String representation of each shape
    String output =
      point.getName() + ": " + point.toString() + "\n" +
      circle.getName() + ": " + circle.toString() + "\n" +
      cylinder.getName() + ": " + cylinder.toString();

    // loop through arrayOfShapes and get name,
    // area and volume of each shape in arrayOfShapes
    for ( int i = 0; i < arrayOfShapes.length; i++ ) {
      output += "\n\n" + arrayOfShapes[ i ].getName() +
        ": " + arrayOfShapes[ i ].toString() +
        "\nArea = " +
        precision2.format( arrayOfShapes[ i ].area() ) +
        "\nVolume = " +
        precision2.format( arrayOfShapes[ i ].volume() );
    }
```

Shape Example: Test Class 3/3

```
 // get name and String representation of each shape
      String output =
         point.getName() + ": " + point.toString() + "\n" +
         circle.getName() + ": " + circle.toString() + "\n" +
         cylinder.getName() + ": " + cylinder.toString();
// loop through arrayOfShapes and get name,
// area and volume of each shape in arrayOfShapes
      for ( int i = 0; i < arrayOfShapes.length; i++ ) {
         output += "\n\n" + arrayOfShapes[ i ].getName() +
            ": " + arrayOfShapes[ i ].toString() + “\nArea = " +
            precision2.format( arrayOfShapes[ i ].area() ) + "\nVolume = " +
            precision2.format( arrayOfShapes[ i ].volume() );
      }
      JOptionPane.showMessageDialog(null,output, "Demonstrating Polymorphism");
      System.exit( 0 );
   }
}  // end class Test
```

Interface Types

- So
- Objects of SimpleRunner has three TYPES:
  - SimpleRunner
  - Runnable and
  - Object

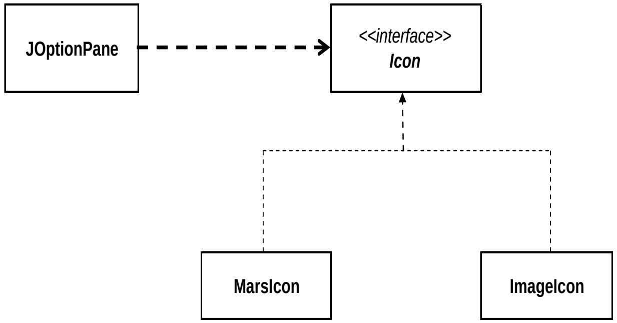


Design Principle 1

- **Program to an interface, not an implementation**

Use abstract classes (and/or interfaces in Java) to define common interfaces for a set of classes Declare variables to be instances of the abstract class not instances of particular classes

- **Benefits of programming to an interface:**
  - **Client classes/objects remain unaware of the classes of objects they use, as long as the objects adhere to the interface the client expects**
  - **Client classes/objects remain unaware of the classes that implement these objects. Clients only know about the abstract classes (or interfaces) that define the interface.**

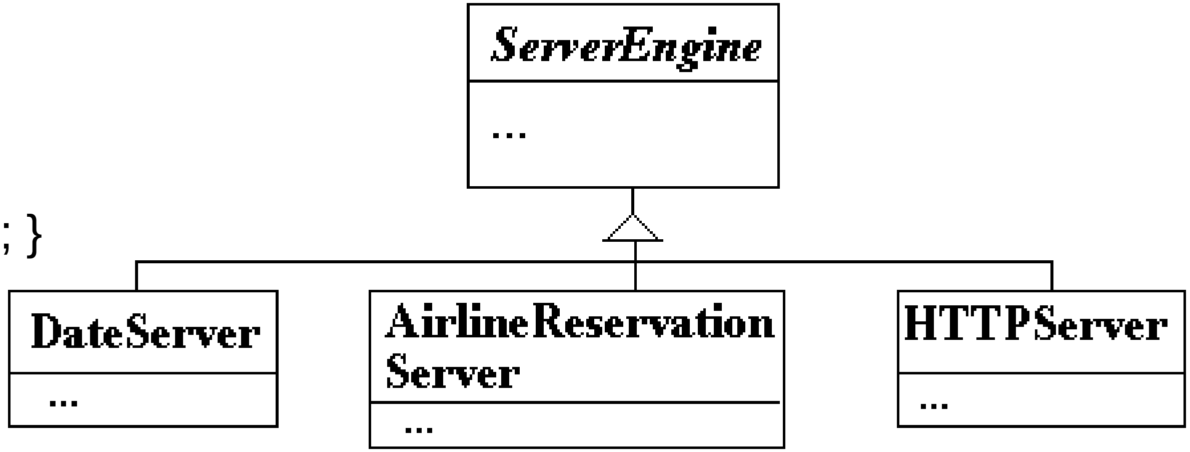


The Icon Example discussed in previous set of slides shows clearly this design Principle

Design Principle 1

- **Programming to an Interface - Example**

```
class A
    {
    DateServer myServer;

    public operation()
        { myServer.someOp(); }
    }

class B
    {
    ServerEngine myServer;

    public operation()
        { myServer.someOp(); }
    }
```


ServerEngine
...
DateServer
...
AirlineReservation Server
...
HTTPServer
...