CS415
INTRODUCTION TO COMPUTER SCIENCE
FALL 2017

8 INTERFACES
CHAPTER 4

LAST TIME

- Inheritance Hierarchies
- Inheritance in Java
- Super class / sub class
- Extending a class
- Overriding inherited methods

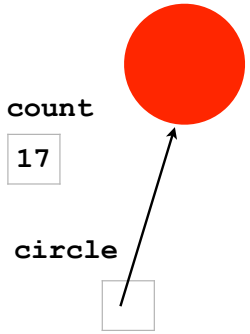
PREVIEW

- Null Pointers.
- Modeling the "acts as" relationship.
- Interface syntax and Implementing interfaces.
- Interfaces in UML.
- Interface Polymorphism.
- Draggable Interface.

MEMORY REVISITED

- Remember:
  - A primitive type variable contains its value.
  - A class type variable contains a reference to its value.

```
int count;
Ellipse circle;

count = 17;
circle = new Elipse( );
```

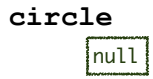


NULL POINTERS

- Remember (draw the memory picture):

```
int count;
Ellipse circle;
```

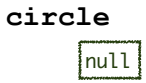
  - A primitive type variable contains its value.
  - A class type variable contains a reference to its value.



- Java automatically initializes variables when they are created.
  - An integer variable is initialized to 0
  - A class type variable is initialized to a null pointer .
- `null` is a Java keyword

NULL POINTER EXCEPTION

- It is an error to access a null pointer




```
Ellipse circle;
circle.setLocation( 20, 30 );

NullPointerException
```

- It might not be obvious if a variable has been initialized.
- We need a way to check.

TESTING FOR NULL POINTER

```
Ellipse circle;
...

if ( circle != null )     // if circle not null
{
     circle.setLocation( 20, 30 );
}
```


IF-THEN-ELSE

- You can also include an else in an if statement
- We'll cover the if statement and associated topics in much more detail in two weeks.

```
Ellipse circle;
...

if ( circle == null )    // test if circle equals null
{
    System.err.println( "Warning: circle is null!" );
}
else
{
    circle.setLocation( 20, 30 );
}
```

MODELING RELATIONSHIPS

- The "has-a" relationship
- When objects are related by a "has-a" relationship we can model the relationship as composite/component.
- For example:
  If in our model a taxi cab has a steering wheel, then a taxi cab will be a composite object with a steering wheel object as a component.

MODELING RELATIONSHIPS

- The "knows-a" relationship
- When objects are related by a "knows-a" relationship we can model the relationship as a peer.
- For example:
  If a Button object needs to tun on a Light object when it is clicked then the button will need to the light as a peer (instance variable)

MODELING RELATIONSHIPS

- The "is-a" relationship
- When objects are related by an "is-a" relationship we can model the relationship by inheritance.
- For example:
  If in our model a taxi cab is a vehicle, then taxi cab objects will be a subclass of the superclass vehicle.

MODELING RELATIONSHIPS

- The "acts-as" relationship
- Sometimes very different objects may share some similar behavior.
- Perhaps a taxi cab can charge a fare and a merry-go-round can charge a fare, but they are probably not related in an inheritance hierarchy.
- They both "act-as" fare chargers.
- How do we model this relationship?

INTERFACE

- An interface is used to model the relationship between different classes whose only commonality is a set of specific shared capabilities.
- That is, they have an "acts-as" relationship
- For example, both TaxiCab and MerryGoRound might implement a Charger interface.
- They both "act as" Chargers, objects that can charge a fare.

INTERFACE

- An Interface specifies a set of capabilities without defining how they will be carried out.
  - It specifies a role that can be played.
- A class can implement an interface by defining all the capabilities specified in the interface.
  - So objects of the class can play the role.

DEFINING A JAVA INTERFACE

```
public interface < interface name >
{
   < method declaration >;
   < method declaration >;

    ...
}
```

- An Interface declares capabilities that an implementing class must have.
- A Java interface consists of a list of method declarations.
- There are no method definitions.

```
public interface Sizable
{
     public void setSize( int w, int h );
     public int getWidth( );
     public int getHeight( );
}
```


IMPLEMENTING A JAVA INTERFACE

```
public interface Sizable
{
     public void setSize( int w, int h );
     public int getWidth( );
     public int getHeight( );
}
```

```
public class Thing implements Sizeable
{
    //instance variables and other methods

    public void setSize( int w, int h )
    {
        // code for setSize
    }

    public int getWidth( )
    {
        // code for getWidth
    }


     public int getHeight( )
    {
        // code for getHeight
    }
}
```

EXAMPLE: AN INTERFACE IN WHEELSUNH

```
/**
 *  This interface models something with a changeable
 *  color. It specifies that all classes implementing
 *  it can have their color set and accessed.
 */

public interface Colorable
{
   // set the color of the implementing object
   public void setColor( java.awt.Color c );

   // get the color of the implementing object
   public java.awt.Color getColor();
}
```


IMPLEMENTING AN INTERFACE

```
public class TaxiCab extends Vehicle implements Colorable
{
    private java.awt.Color myColor;

    // other instance variables and methods

    public void setColor( java.awt.Color c )
    {
        MyColor = c;
    }

    public java.awt.Color getColor()
    {
        return MyColor;
    }
}
```

```
public interface Colorable
{
   public void setColor( java.awt.Color c );
   public java.awt.Color getColor();
}
```

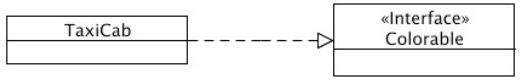

IMPLEMENTING AN INTERFACE

- We can write the setColor and getColor methods with or without implementing the interface.
- The interface enforces consistency:
- all classes implementing the interface will have all the interface methods with the same signatures and return types.

```
public interface Colorable
{
   // set the color of the implementing object
   public void setColor( java.awt.Color c );

   // get the color of the implementing object
   public java.awt.Color getColor();
}
```


UML FOR INTERFACES

- Example: TaxiCab implements Colorable
  - Dotted line
  - Hollow triangular arrow points to Interface
  - << Interface >> Label



INTERFACE POLYMORPHISM

- Interfaces can be used as types for variables and parameters, just like class types and primitive types.
- You can declare an instance or local variable as an interface.
- And this can reference any class that implements Colorable:

```
TaxiCab    cab = new TaxiCab();
Colorable theColorable;
theColorable = cab;
```


INTERFACE POLYMORPHISM

- Suppose we want methods that will paint TaxiCabs and Rectangles.
- We could write two methods:

```
public void paint( TaxiCab theCab )
{
    .....
}

public void paint( Rectangle theRectangle )
{
    .....
}
```

- But since TaxiCabs and Rectangles are both Colorable we could write one method.

```
public void paint( Colorable theColorable )
{
   ...
}
```


INTERFACE POLYMORPHISM

```
import wheelsunh.users.*;
import java.awt.Color;
/**
 * A paint shop that can apply a random Color to
 * any object that is Colorable.
 */

public class PaintShop
{
    // constructors and other methods
    //  including:  private Color getRandomColor()

    public void randomPaintJob( Colorable item )
    {
       Color rColor = getRandomColor();
       item.setColor( rColor );
    }
}
```


INTERFACE POLYMORPHISM

```
public class PaintShop
{
    ...
    public void randomPaintJob( Colorable item )
    {
       Color rColor = getRandomColor();
       item.setColor( rColor );
    }
}
```

```
public class PaintApp
{
    private TaxiCab cab;
    private Rectangle rect;
    private PaintShop shop;

    public PaintApp()
    {
        cab = new TaxiCab();
        rect = new Rectangle();
        shop = new PaintShop();

        shop.randomPaintJob( cab );
        shop.randomPaintJob( rect );
    }
    ...
}
```

MULTIPLE INTERFACES

- In Java a class can extend only one class.
- But, a class can implement any number of interfaces.
- A TaxiCab, for example, might *extend* Vehicle class and *implement* Colorable and Movable interfaces.

```
public class TaxiCab extends Vehicle implements Colorable, Movable
```


INTERFACE VS. CLASS

Interface	Class
Models a role; defines a set of responsibilities	Models an object with properties and capabilities
Factors out common capabilities of dissimilar objects	Factors out common properties and capabilities of similar objects
Declares, but does not define, methods	Declares methods and may define some or all of them
A class can implement multiple interfaces	A class can extend only one superclass

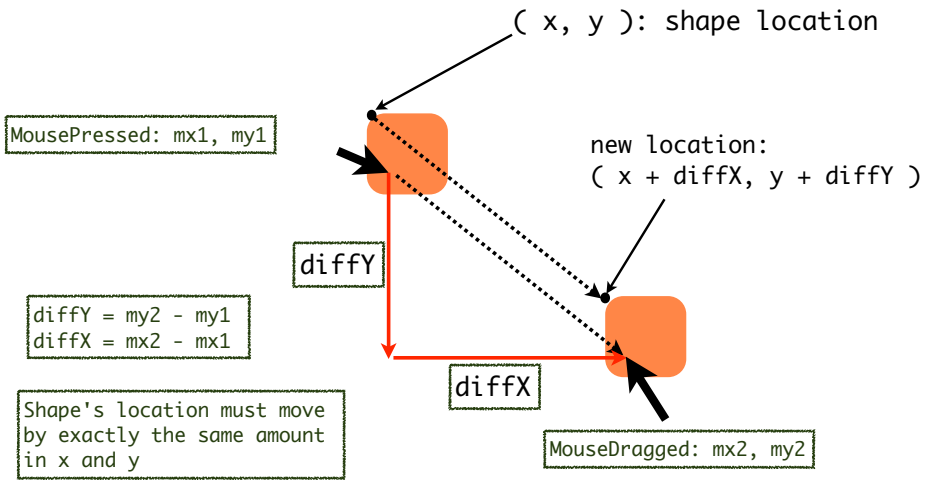
DRAGGABLE OBJECTS

- A draggable graphics object should :
  - respond to mouse events
  - change its position
  - change its color while it is being moved (for user feedback).

IMPLEMENTING DRAGGABLE

```
import wheelsunh.users.*;
import java.awt.Color;
import java.awt.Point;
import java.awt.event.*;

public class Block extends Rectangle
{
    private Point lastMousePosition;

   // constructors and other methods

   // inherited  public void setColor( Color aColor )
   // inherited  public void setLocation( int x, int y )

   public void mousePressed( MouseEvent e )
   {
      ...
   }

   public void mouseDragged( MouseEvent e )
   {
      ...
   }

   public void mouseReleased( MouseEvent e )
   {
     ...
   }
}
```

HOW DO WE DRAG?



IMPLEMENTING DRAGGABLE

- MousePressed: to start the drag get the initial position of the mouse and sets the color to blue.
- MouseReleased: to end the drag set the color back to red.

```
public void mousePressed( MouseEvent e )
{
   lastMousePosition = e.getPoint();
   this.setColor( java.awt.Color.BLUE );
}

public void mouseReleased( MouseEvent e )
{
    this.setColor( Color.RED );
}
```


IMPLEMENTING DRAGGABLE

- MouseDragged gets the change in the mouse position
- Then it moves the object by the change.

```
public void mouseDragged( MouseEvent e )
 {
     Point currentPoint = e.getPoint();
     int diffX = currentPoint.x - lastMousePosition.x;
     int diffY = currentPoint.y - lastMousePosition.y;
     setLocation( getLocation().x + diffX, getLocation().y + diffY );
     lastMousePosition = currentPoint;
  }
```



```
import wheelsunh.users.*;
import java.awt.Color;
import java.awt.Point;
import java.awt.event.*;
public class Block extends Rectangle
{
   private Point lastMousePosition;

   public Block()
   {
   }

   // inherited  public void setColor( Color aColor )
   // inherited  public void setLocation( int x, int y )

   public void mousePressed( MouseEvent e )
   {
      lastMousePosition = e.getPoint();
      this.setColor( Color.blue );
   }

   public void mouseDragged( MouseEvent e )
   {
       Point currentPoint = e.getPoint();
       int diffX = currentPoint.x - lastMousePosition.x;
       int diffY = currentPoint.y - lastMousePosition.y;
       setLocation( getLocation().x + diffX, getLocation().y + diffY );
       lastMousePosition = currentPoint;
   }

   public void mouseReleased( MouseEvent e )
   {
       this.setColor( Color.RED );
   }
}
```

REVIEW

- Modeling the "acts as" relationship.
- Interface syntax.
- Implementing interfaces.
- Interfaces in UML
- Interface Polymorphism
- Draggable Interface

NEXT TIME

- Inheritance Polymorphism
- Declared Type/ Actual Type
- Method Resolution

- Read Chapter 5