CSE 2102: Introduction to Software Engineering

Object Oriented Design

Greg Johnson

Computer Science & Engineering Department
University of Connecticut

August 26, 2019

Table of contents

Methods Review

- Class capabilities provided via methods
  - message gets sent to object
  - code gets executed
  - response sent back on completion
- Public methods can be used by any function which defines an object of the class.
- Private methods can only be invoked by the methods of that class.

Parameters

- Allow us to generalize the methods
  - provide specific information
  - provide specific instances on which to work
- May be class instances, of either the same or different class.
- Are references to objects, and copies of primitive types(int).

Formal Parameters

- Formal parameters are defined in the method header.
- Formal parameters are working names for details to be filled in later.
- Formal parameters (syntax)
  - list of declarations separated by commas and enclosed by parentheses
  - each declaration consists of type name

Actual Parameters

- Actual parameters are the values you fill in when you actually use the method
- Actual parameters must match formal parameters
  - in type,
  - position,
  - and number,
  - but not in name.

Some Examples

- `public void setLocation (int x, int y){...}`
  - `int` is a type
  - `x` and `y` are formal parameters
- `public void setColor(java.awt.Color color){...}`
  - `java.awt.Color` is a type
  - color is a formal parameter

Parameter passing

On method invocation

- Formal parameters (in method) are set to values of actual parameters (in call).
- non-primitive data type parameters
  - method can alter it
  - mutating attributes of an object which is a parameter will mutate the attributes of the original object
  - can not change the parameter itself (can not create a new object for that parameter.

Signature

The signature for a method consists of:

- the name of the method
- the class and order of the parameters

When a message is sent/a method is invoked, it must match the signature

- have the same name
- arguments match the data type(class) and order of the parameters.

Example

Method Signature

- `public void setLocation(int x, int y)`
  - method name `setLocation`
  - parameter names `x` and `y`
  - parameter types `int` and `int`

Method Invocation

- `_myBall.setLocation(100, 200);`
  - method name `setLocation`
  - values of type `int`, 100 and 200

Overloading of Methods

- Overloading occurs when there is more than one method defined with the same name but different implementations.
- If the overloaded methods are in the same class they must have different signatures.
- The methods may have the same signature if they are in different classes.

Return Values and Methods

- The return type is the second entity specified in the method header.
  - specifies the data type of the value returned
- Allows access to the values of instance variables of an object.
- `public java.awt.Color getColor(){...}`
  - `java.awt.Color` is the return type
- `ballColor = myBall.getColor();`
  - where ballColor is a declared to be a java.awt.Color

Types of Methods

Constructors

- Initialize objects when declared
- may have both default constructor(no parameters) and parameterized constructors

Accessors

- Return the value of a data member - “get”

Mutators

- Modify a data member - “set”

Review of Constructors

- Purpose of constructor is to initialize the instance variables.
- Constructors have no (stated) return type
- Constructors share the same name with the class.

Recipes

- "Appropriate laziness"
- recognize that there are certain ways of doing things that recur. Deal with these in the same way
- Easier to write code
- Easier to understand code

Component Property Recipe

- When class has parts or components
  - components - specified by individual instance variables
  - components instantiated in constructor

```
public class <className> {
    private <type1> _<component1>;
    . . .
    // constructor of class
    public <className> (<parameters>) {
    _<component1> = new <type1>();
    }
}
```

Attributes

- Attributes are simple objects or primitive types (i.e. `int`)
- Attributes are initialized either to values passed to the constructor or to arbitrary values.

Association Recipe

- When an object is associated with an independent instance - it needs to know about that instance.
  - Example: A crow lives in a particular tree. The tree object is a peer object or associate of the crow class
  - peer objects (associates) are specified as instance variables
  - peer objects set using parameter in constructor

Association Recipe Example

```
public class Crow {
    private Tree _homeTree;// instance variable
    . . .
    // constructor for Crow
    public Crow(Tree home) {
        _homeTree = home;
        . . .
    }
}
```

Syntax convention: the parameter name is different from the instance variable name.

Delegation Recipe

- Objects perform actions or solve problems by dividing up the problem and delegating actions to the components.

Example:

```
public void spreadAlarm() {
    _wings.flap();
    _beak.makeSound(alarm);
}
```

this

- this() is equivalent to invoking the constructor for the same class as the message
- Example:

```
public class Ellipse {
    private java.awt.Color _color;
    . . .
    public Ellipse (java.awt.Color color) {
        _color = color;
    }
    public Ellipse () {
        this(java.awt.Color.RED); // invokes Ellipse constr
    }
    . . .
}
```

Default Value for Constructor Recipe

```
public class <className> {
    private <type> _<instanceVar>;
    . . .
    public <className> (<type> my<Type>) {
        _<instanceVar> = my<type>;
    }
    public <className> () {
        this(defaultValue);
    }
. . .
}
```

Recipes - Accessor and Mutator Methods

A recipe is a standard way to do something.

- Accessor methods are generally named "get"
- Accessor recipe
  `public <attributeType> get<attributeName>()`
- Mutator methods are generally named "set"
- Mutator recipe
  `public void set<attributeName>(<attributeType> var)`

Other Accessor and Mutators for Ellipse

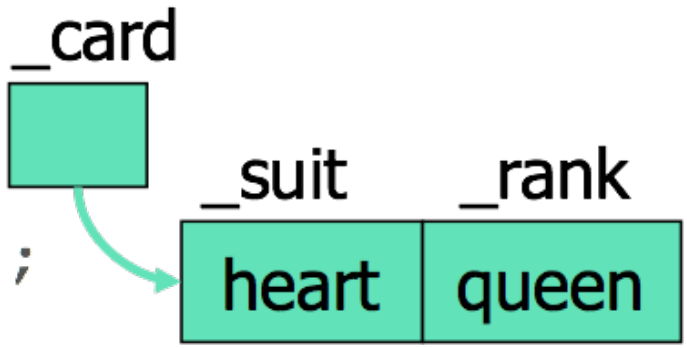
- `public int getWidth();`
- `public void setWidth(int w);`
- `public int getHeight();`
- `public void setHeight(int h);`

Java Objects and References

When you declare an object

- sets up a reference to object
- the actual object does not exist until it is instantiated.

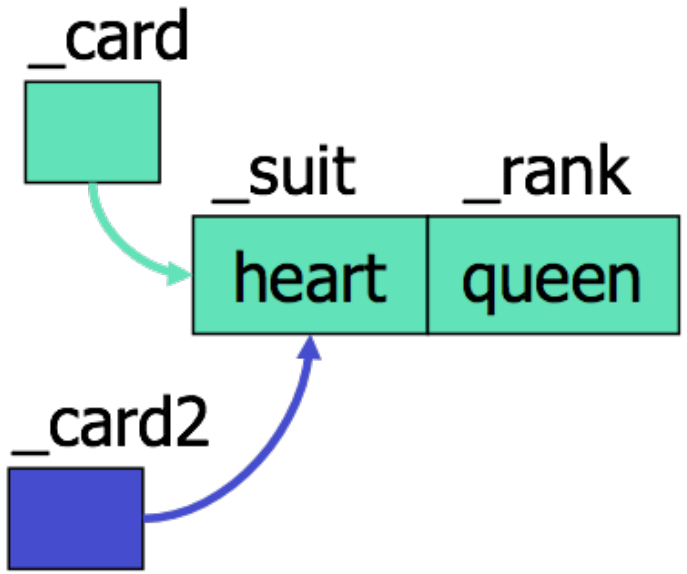
```
Card _card;
_card = new Card('H', 'Q');
```



Java Objects and References

Now we declare another card, and assign _card2 to it.

```
Card _card;
_card = new Card('H', 'Q');
Card _card2;
_card2 = _card;
```



Any changes to _card happen to _card2 as well, and changes to _card2 change _card, since they refer to the same object.

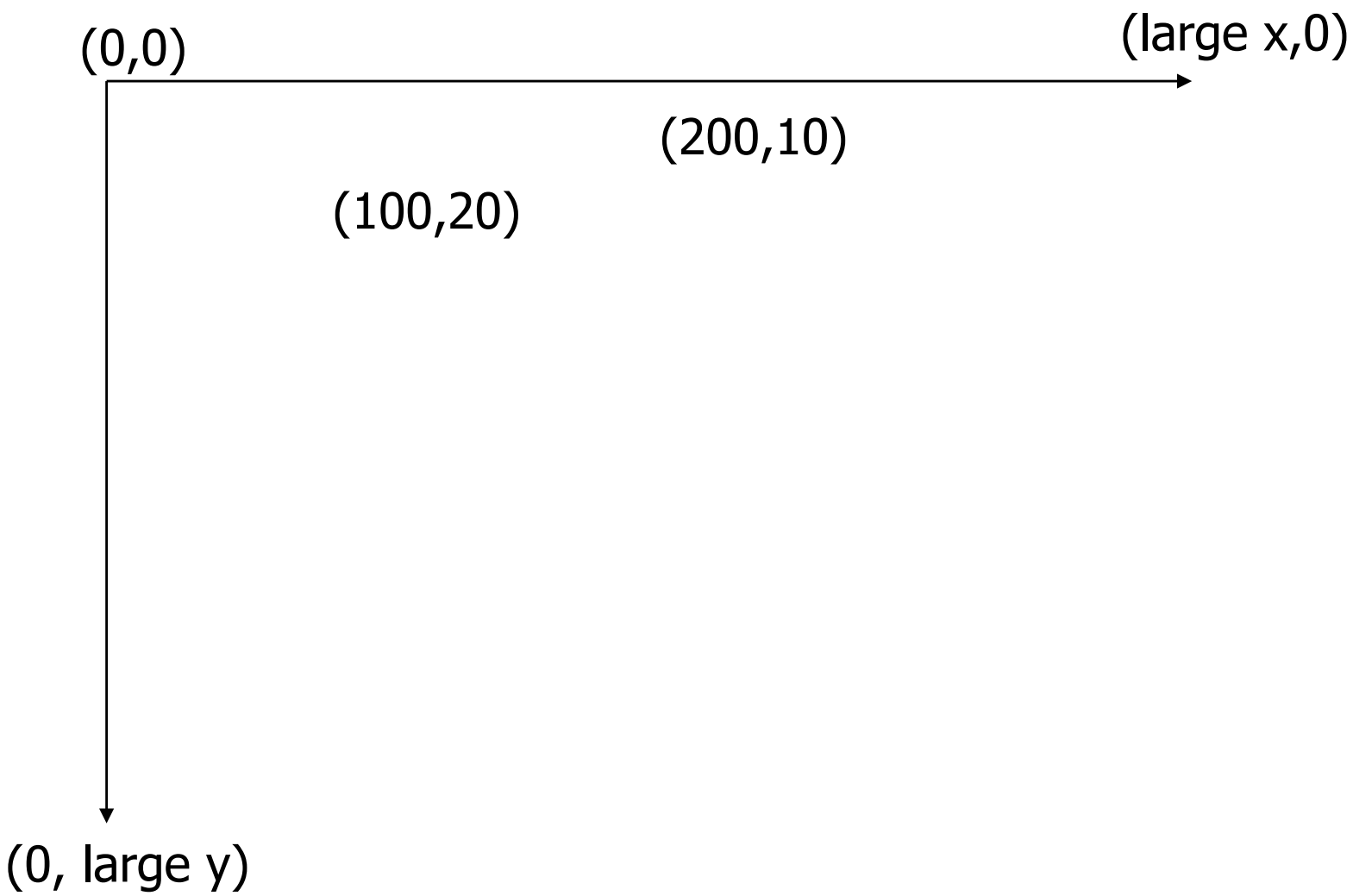
Kinds of Variables

- Instance Variables
  - data members of a class, components and peers
  - Style - start with _ followed by lowercase letter, capital letter for each new word
- Parameters
  - information passed to a method
  - Style - start with lowercase letter, capital letter for each new word
- Local Variables
  - declared within a method temporary storage
  - Style - start with lowercase letter, capital letter for each new word

Scope and Lifetime

- Instance Variables
  - **Scope** - At end of class in which declared
  - **Lifetime** - While object exists
  - **When Used** - To model characteristic used in more than one method
- Parameters
  - **Scope** - At end of method in which declared
  - **Lifetime** - During response to particular message
  - **When Used** - To model something obtained from outside
- Local Variables
  - **Scope** - At end of method in which declared
  - **Lifetime** - During response to particular message
  - **When Used** - To model something internally for that method only.

Absolute Positioning in a Frame



Relative Positioning

- Use a point (x,y) for the position of the composite object. Then the position of all component objects are specified as offsets from point (x,y).
- For example if the position of the composite object is at (200, 250), a component can be at position (350, 275). For example, use `_frontWheel.setLocation(x+150,y+25);`
- Now, if the position of the vehicle object is changed, the position of the wheel will also be changed appropriately.

Why Inheritance?

- Consider the graphics classes you (may) have worked with: Rectangles, Ellipses, ConversationBubbles, etc.
- There are many things which they share in common.
  - Anchor point
  - Length and width
  - Color
- Each has a number of things they do not
  - The particular shape of the graphic
  - The text in the conversation bubble

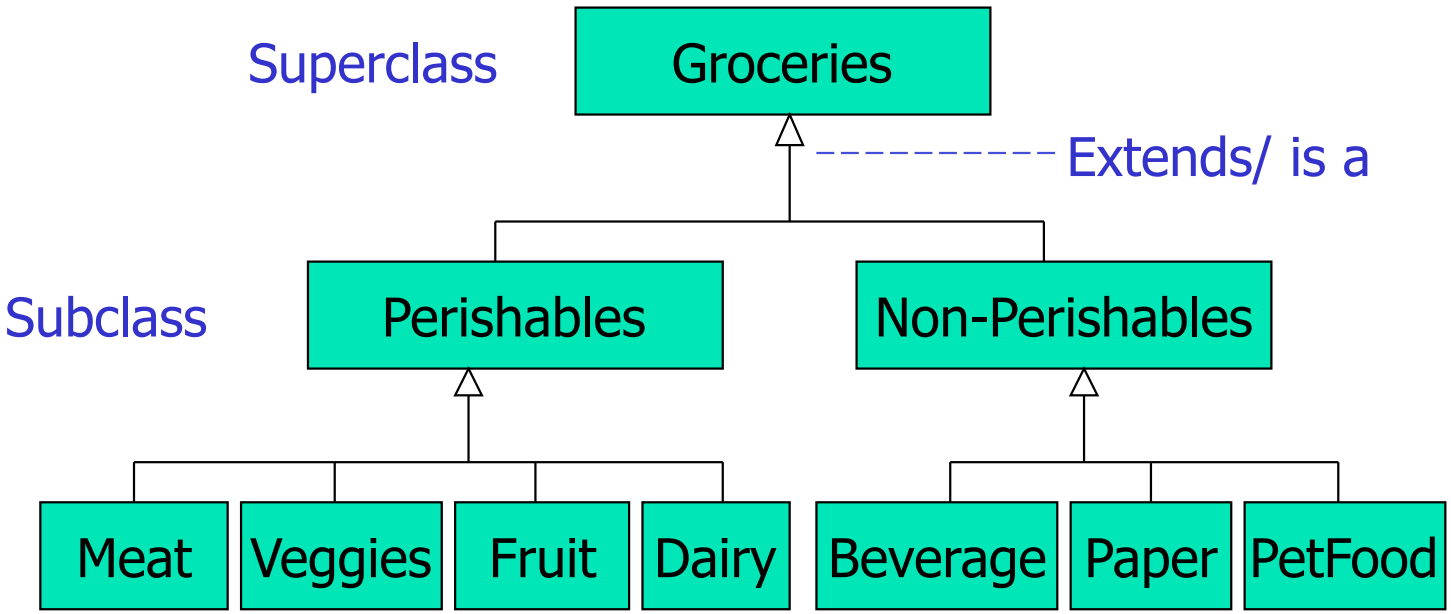
Inheritance

- Allows for the common attributes and capabilities to be shared.
- Builds a hierarchy of classes
- Superclass - Given the common attributes and capabilities
- Subclass - Inherits the common attributes and capabilities and can add to them or modify them for specialization

Relationships

- **Has A** - contains
  - component, Crow has a wing
- **Knows its** - has knowledge of , related to
  - peer - An employee knows his boss
- **Is A** specialization relationship
  - inheritance - Meat is a perishable item

Diagram of Simple Inheritance

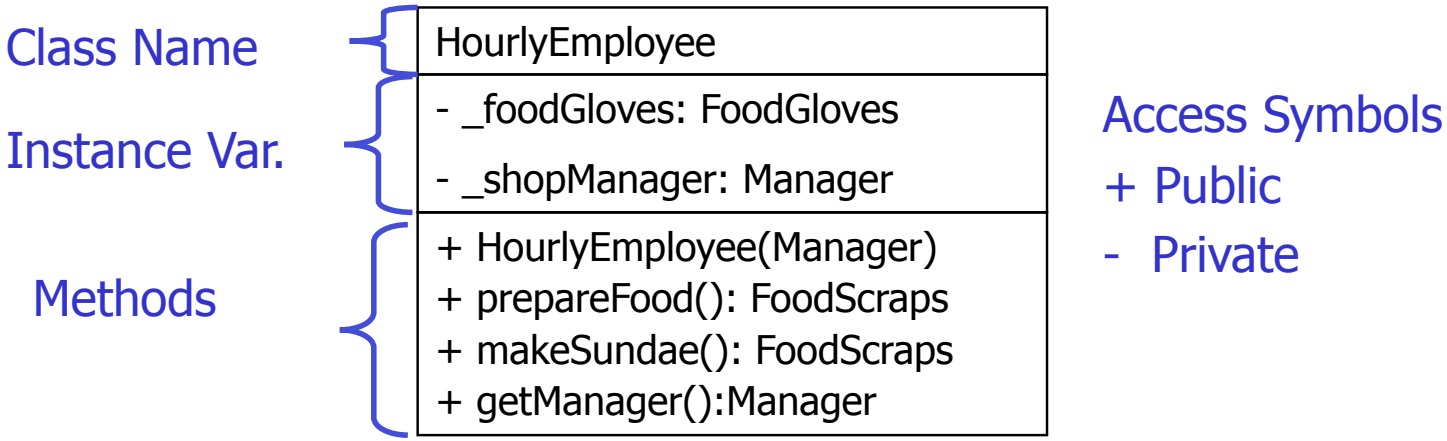


- A superclass can have many subclasses.
- A subclass has only one superclass

Java Syntax and Examples

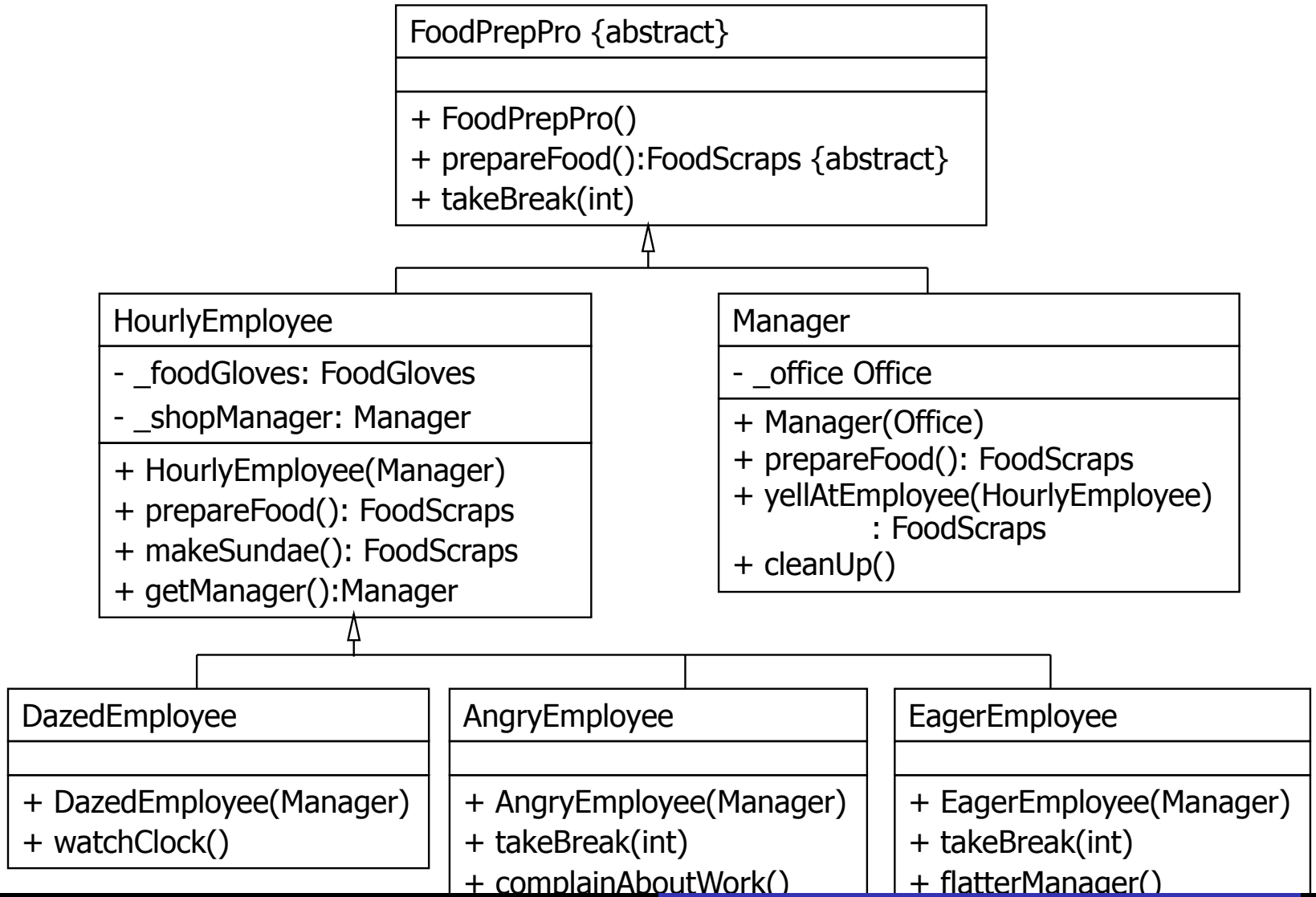
- `public class <subclass> extends <superclass>`
- `public class MyApp extends javax.swing.JFrame { }`
- `public class Perishable extends Groceries { }`
- `public class PetFood extends NonPerishable { }`

Recall: UML Class Diagram



- Instance Variables
  `<AccessSymbol> <VariableName> : <Class>`
- Method
  `<AccessSymbol> <MethodName>(<ParameterTypes>) : <Return Type>`

Example: Food Prep Professional



Returning to Example

- In example: `HourlyEmployee` and `Manager` must implement prepareFood method.
- Instances of `HourlyEmployee`, `Manager`, `DazedEmployee`, `AngryEmployee`, `EagerEmployee` are allowed.

Example of Abstract classes

FoodPrepPro is an abstract class, therefore we can not instantiate a FoodPrepPro, but rather must specify type of FoodPrepPro. We can however declare a reference to a FoodPrepPro.

- `FoodPrepPro samantha; // is valid`
- `samantha = new FoodPrepPro(); // is not valid`
- `samantha = new Manager(); // is valid`
- `Manager sarah; // is valid`
- `sarah = new Manager(); // is valid`
- `HourlyEmployee waiter; // is valid`
- `waiter = new HourlyEmployee(sarah); // is valid`

HourlyEmployee

- Inherits `takeBreak(int)` method from `FoodPrepPro` class
- Has its own constructor with parameter `Manager`
- allows it to set its private variable `_shopManager`
- Implements `prepareFood()` method
- therefore, instances of `HourlyEmployee` allowed
- Has additional capability `makeSundae()` method

DazedEmployee

- Inherits
  - `takeBreak(int)` from `FoodPrepPro` class
  - `prepareFood()` from `HourlyEmployee` class
  - `makeSundae()` from `HourlyEmployee` class
- Inherits but does not have access to
  - `_foodGloves` and `_shopManager` since they are private, but can use methods of `HourlyEmployee` which do have access to them
- Implements its own constructor, and `watchClock()` method.
- Instances of `DazedEmployee` are allowed.

AngryEmployee

- Inherits
  - `prepareFood()` from `HourlyEmployee`
  - `makeSundae()` from `HourlyEmployee`
- Inherits but does not have access to
  - `foodGloves` and `_shopManager` since they are private, but can use methods of `HourlyEmployee` which do have access to them
- Overrides method `takeBreak()`
- Additionally implements constructor and `complainAboutWork()` methods.
- Instances of `AngryEmployee` are allowed.

EagerEmployee

- Inherits
  - `prepareFood()` from `HourlyEmployee`
  - `makeSundae()` from `HourlyEmployee`
- Inherits but does not have access to
  - `_foodGloves` and `_shopManager` since they are private, but can use methods of `HourlyEmployee` which do have access to them
- Overrides method `takeBreak()` Additionally implements constructor and `flatterManager()` methods. Instances of `EagerEmployee` are allowed.

`super()`

- `super()` invokes the constructor of the superclass
- It will have parameters if the superclass constructor has parameters.
- Must be first statement in constructor of subclass. If not explicit, implicit invocation is made with no parameters. Error, if no default constructor for superclass is invoked.

`super.`

- Use `super.` to access method of superclass instead of your own method by that name.
- Example: Inside of implementation of `takeBreak(int)` of `AngryEmployee`

```
super.takeBreak(2 * time);
```

Java Implementation of FoodPrepPro

```
public abstract class FoodPrepPro {
    // default constructor
    public FoodPrepPro(){ }

    // other methods
    public abstract FoodScraps prepareFood();
    public void takeBreak(int time){ ... }
}
```

Java Implementation of HourlyEmployee

```
public class HourlyEmployee extends FoodPrepPro {
    // instance variables
    private FoodGloves _foodGloves;
    private Manager _shopManager;
    // constructor, uses association pattern
    public HourlyEmployee(Manager mngr)  {
        super();// opt, invokes constructor of FoodPrepPro
        _shopManager = mngr;
        _foodGloves = new FoodGloves();
    }
    // implementation of other methods
    public FoodScraps prepareFood()  {
        FoodScraps yourFood = new FoodScraps();
        return yourFood;
    }
    public FoodScraps makeSundae()  {
        FoodScraps sundae = new FoodScraps();
        return sundae;
    }
```

Java Implementation of Manager

```
public class Manager extends FoodPrepPro  {
    // instance variables
    private Office _office;
    // constructor
    public Manager(Office office) {
        super();
        _office = office;
    }
    // other methods
    public void yellAtEmployee()  {  }
    public void cleanUp() {   }
    // override abstract prepareFood of FoodPrepPro
    public FoodScraps prepareFood() {
        private FoodScraps food;
        food = _employee.prepareFood();
        return food;
    }
```

Java Implementation of AngryEmployee

```
public class AngryEmployee extends Hourly Employee {
    // constructor
    public AngryEmployee(Manager mngr)  {
        super(mngr);  // invokes constructor for HourlyEmpl
    }
    // other methods
    public void complainAboutWork()  {  }
    public void takeBreak(int time)  {
        // invokes takeBreak of HourlyEmployee
        // which is the takeBreak inherited from
        // FoodPrepPro
        super.takeBreak(2 * time);
    }
}
```

Java Implementation of EagerEmployee

```
public class EagerEmployee extends Hourly Employee  {
    // constructor
    public EagerEmployee(Manager mngr)  {
        super(mngr);  // invokes constructor for HourlyEmpl
    }
    // other methods
    public void flatterManager()  {
        Manager myBoss;
// local variable
        // HourlyEmployee must have getManager method
        myBoss = super.getManager();
        myBoss.receiveFlatteryFrom(this);
    }
    public void takeBreak(int time)  {
        // invokes takeBreak of HourlyEmployee
        // which is the takeBreak inherited from
        // FoodPrepPro
        super.takeBreak(time/2);
```

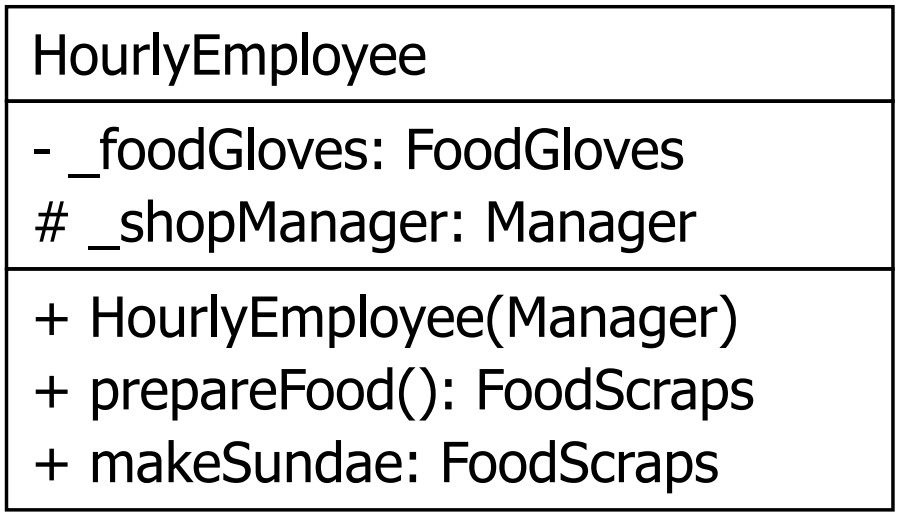
Why use Private Instance Variables?

- Good information hiding.
- Can provide controlled access through "get" method in superclass to its subclasses.
- It can be used in superclass methods without subclass needing to know about it.
- For example, if `HourlyEmployee` had a `flatterManager` method all `EagerEmployee` would need to do was invoke it.

Another Approach to the Problem

- We don't want to give everyone access to the Manager of an `HourlyEmployee`, but it would be more convenient if the subclasses of `HourlyEmployee` had access to it.
- Protected - Makes an instance variable or method available to the subclasses of the class where it is declared, but not outside of the class hierarchy.
- # is the symbol used in UML to specify protected access.

Making _shopManager Protected




```
public class HourlyEmployee extends FoodPrepPro {
    private FoodGloves _foodGloves;
    protected Manager _shopManager;
    public HourlyEmployee(Manager mngr) {   }
    public FoodScraps prepareFood() {   }
    public FoodScraps makeSundae() {   }
}
```

Changes to EagerEmployee

```
public class EagerEmployee extends Hourly Employee  {
    // constructor
    public EagerEmployee(Manager mngr) {
        super(mngr);
    }
    // other methods
    public void flatterManager()  {
        _shopManager.receiveFlatteryFrom(this);
}
```

Yet Another Approach

- We can make the `getManager()` method protected, and keep the `_shopManager` private.
- Now, subclasses that dont need to know access to the `_shopManager` dont have access to it, but if they need it they can get it through the `getManager()` method.
- The outside world is not given access to the `_shopManager`, since they can not use the `getManager()` method.

Revised flatterManager

```
public void flatterManager()  {
    Manager myBoss;                 // local variable
    // HourlyEmployee must have getManager method
    myBoss = super.getManager();
    myBoss.receiveFlatteryFrom(this);
}
```

Things an Object Can Access:

- Its methods and instance variables
- Public instance variables and methods of any class
- Protected methods and instance variables of its superclasses

Things an Object Can't Access:

- Private instance variables and methods of any other class, including its superclass.
- Protected instance variables of classes other than those of its superclass.

Method Resolution

Which method gets invoked?

- Look in the class of the object invoking the method, if it has a method with the signature use it.
- Otherwise look in its immediate superclass.
- Continue up the hierarchy until a method of the appropriate signature is found. If not is found, report failure (compile error)

Example of Method Resolution

```
Manager mike = new Manager();
HourlyEmployee waitStaff= new HourlyEmployee(mike);
DazedEmployee daniel = new DazedEmployee(mike);
AngryEmployee alan = new AngryEmployee(mike);
EagerEmployee ellen = new EagerEmployee(mike);

mike.takeBreak(2);              // uses FoodPrepPro takeBreak
waitStaff.takeBreak(2);// uses FoodPrepPro takeBreak
daniel.takeBreak(2);    // uses FoodPrepPro takeBreak
alan.takeBreak(2);            // uses AngryEmployee takeBreak
ellen.takeBreak(2);         // uses EagerEmployee takeBreak
```

Inheritance

- Allows us to keep attributes and methods at higher abstraction level
  - more compact code
- Abstract methods are "inherited" as well, since they force subclasses to implement them.
- Overriding methods allows subclass to define capabilities which are different from parent, but with the same signature.

Grouping Mechanisms

- **Classes** - Group objects that have the same attributes and capabilities
- **Inheritance** - Group classes into a hierarchy of increasing complexity where the "is a" relationship exists between the different levels.
- **Interfaces** - Group classes based on common capabilities or roles, when there is little other similarity between them. Can be described as an "acts as" relationship.

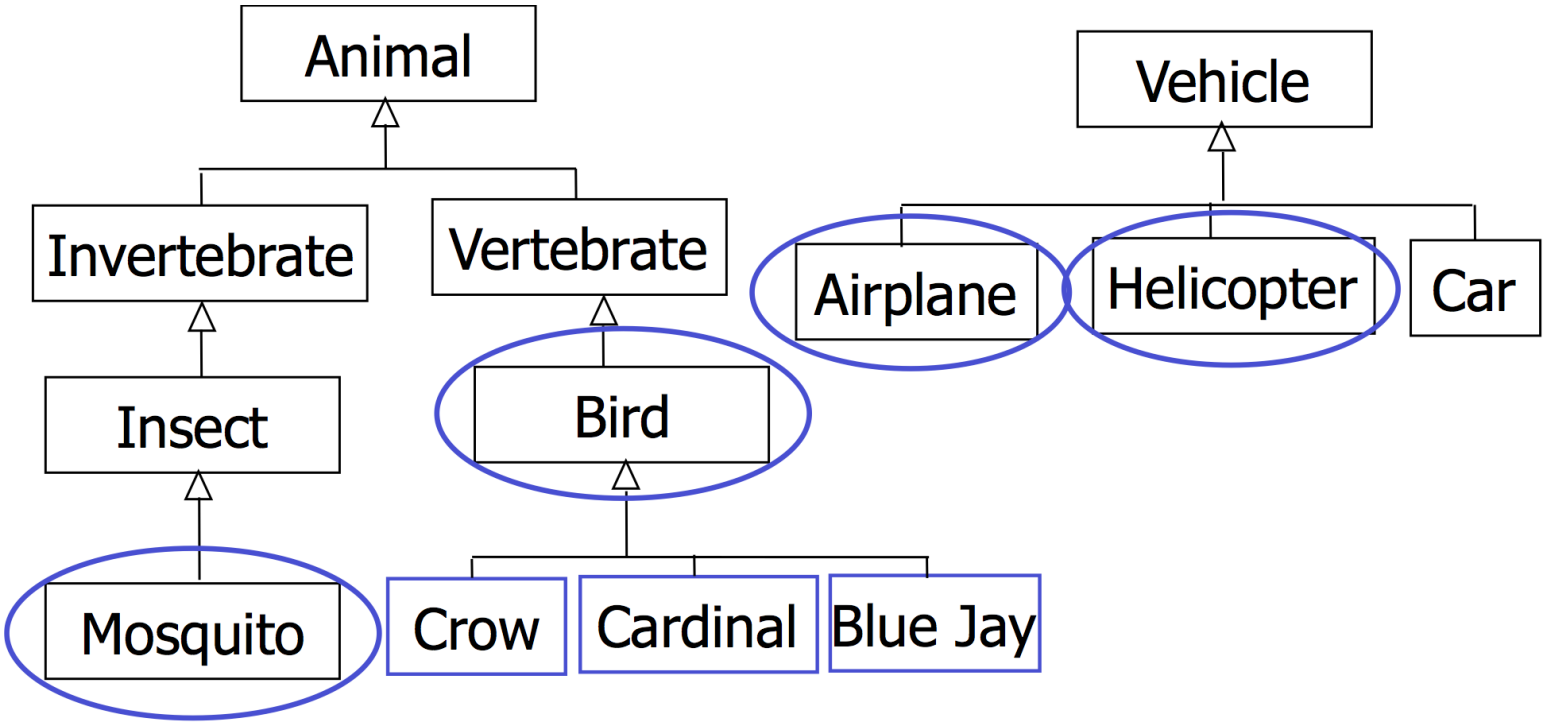
Java Interfaces

- Model *roles* - common capabilities
  Ex: ability to fly (Airplane, Bird, Mosquito)
- Interfaces specify list of abstract methods which must be implemented by the classes that implement the interface.
- Interfaces contain no instance variables, but may contain constants.

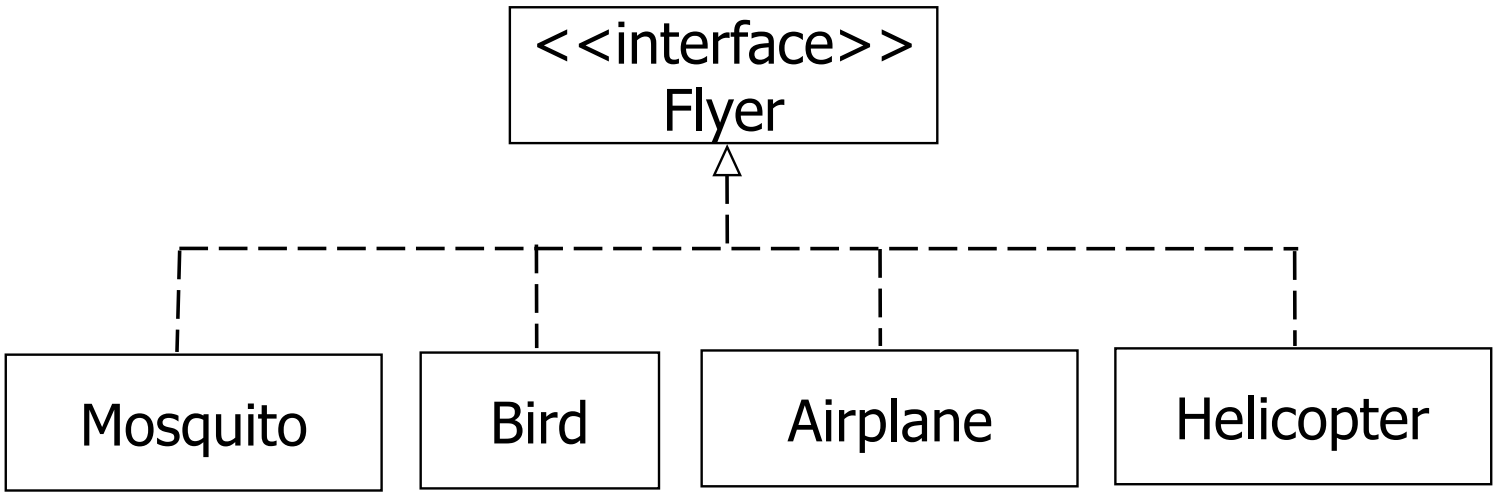
Naming of Interfaces

- Since interfaces model roles or capabilities their names often end in er or able.
- For example:
  - Mover
  - Flyer
  - Shaker
  - Colorable
  - Mouseable
  - Moveable

Airplane, Helicopter,Bird, Mosquito



UML Class Diagram (Interfaces)



The Interface Guarantee

- If a non-abstract class `implements` an interface it has concrete methods for all methods listed in the interface.
- If a class implements an interface and does not implement all methods of the interface it is an abstract class by definition.
- It can either define the methods itself or inherit them from its superclass.

Creating An Interface In Java

```
public interface <interfaceName>
{
        <method declarations>
}
```

Each method declaration consists of:

```
<access> abstract <returnType> <methodName> (<parameterList>);
```

Example: Flyer Interface

```
public interface Flyer
{
public abstract void fly (Location start, Location end);
public abstract void takeOff(Location start);
public abstract void land(Location end);
}
```

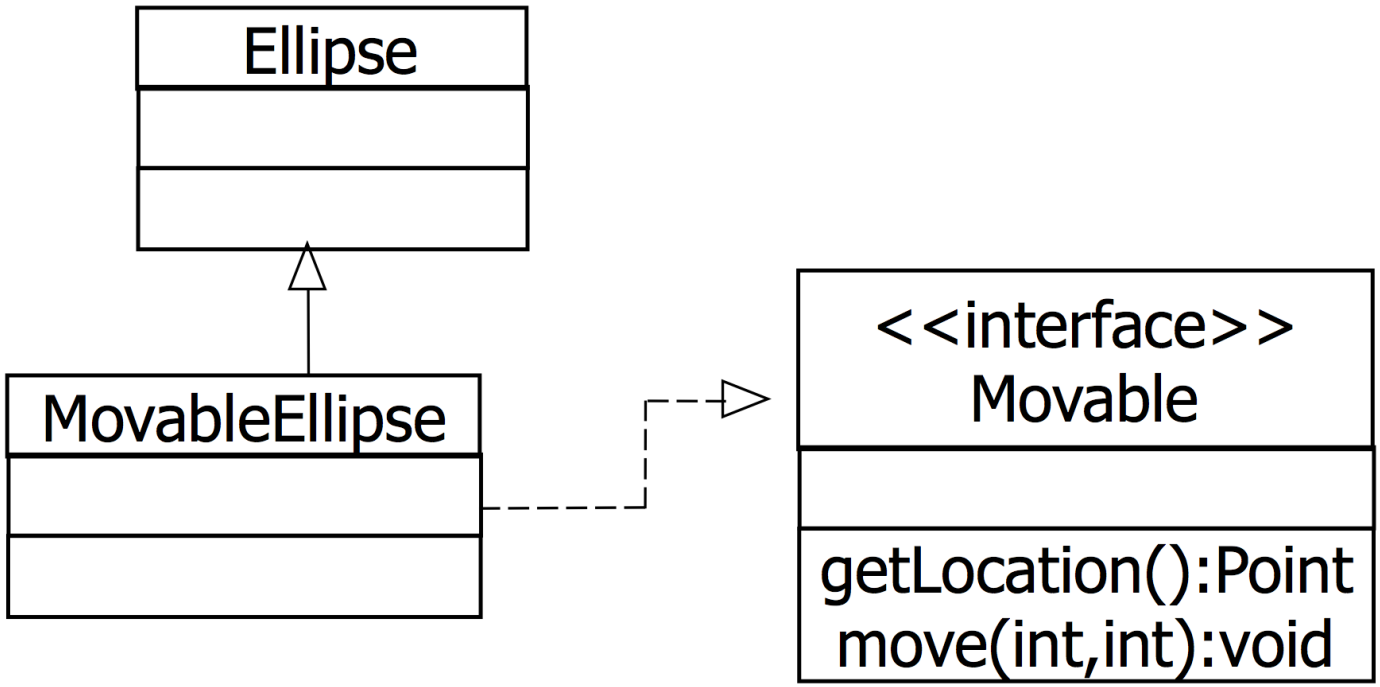
Classes and Interfaces

```
public class<className> implements<interfaceName>{...}
```

Examples:

```
public class Airplane extends Vehicle implements Flyer{...}
public class Bird extends Vertebrate implements Flyer{...}
public class Mosquito extends Insect implements Flyer{...}
```

Example UML Class Diagram



Another Example

```
public interface Movable
{
  public abstract Point getLocation();
  public abstract void move (int dx, int dy);
}
```

Another Example: Movable Ellipse

```
public class MovableEllipse extends Ellipse implements Movable
{
. . .
  public Point getLocation(){
    return new Point(getXLocation(), getYLocation());
  }
  public void move (int dx, int dy){
    setXLocation(getXLocation()+dx);
    setYLocation(getYLocation()+dy);
  }
. . .
}
```

A Class Can Implement More Than One Interface

If we have an interface Draggable we can have our movableEllipse also implement Draggable

```
public interface Draggable
  {
    public abstract void mousePressed (MouseEvent e);
        public abstract void mouseDragged (MouseEvent e);
        public abstract void mouseReleased (MouseEvent e);
        }

public class MovableEllipse extends Ellipse
                            implements Draggable, Movable
{     }
```

where it needs to implements the methods needed by Movable and Draggable.

Interfaces vs. Abstract Classes

Similarities

- both have abstract methods
- both can not be instantiated

Differences

- Interfaces instance variables are all public static final (i.e. constants) and shouldn't be used.
- Interfaces have only abstract methods, i.e. no concrete methods
- Interfaces model roles, while abstract classes model a natural category of objects.

Interfaces vs. Abstract Classes

Interface

- Models role; defines set of responsibilities
- Factors out common capabilities of potentially dissimilar objects
- Declares, not defines methods
- Class can implement multiple interfaces

Abstract Class

- Models object with properties and capabilities
- Factors out common properties and capabilities of similar objects
- Declares methods and may define some of them
- Class can extend only one superclass

How to choose?

- Classes should model object with properties and capabilities. Interfaces model roles.
- A class may have one superclass, but zero or more interfaces
- Interfaces provide abstract description of object based on some behavior. Models only that aspect of the object.

Practice

Suppose you want to model that the classes Cod, Trout, and Goldfish are all fish.

- Would you use an interface or inheritance? Why?

Suppose you want to model that the classes Frog, Trout, and Whale are all swimmers.

- Would you use an interface or inheritance? Why?

Box class

- Boxes can both hold stuff and can be held by other things/people.
- Given interfaces Holder and Holdable, a box would implement both.
- Holders: can hold and release a Holdable thing.
- We are defining the type of the things held and released by a Holder as an interface!!
- Holdable: can be held by a Holder.

The Holder and Holdable Interfaces

```
public interface Holder
{       public abstract void add(Holdable toBeHeld);
        public abstract Holdable release(Holdable toBeReleased);
}

public interface Holdable
{       public abstract Weight getWeight();
        public abstract Dimensions getDimensions();
}
```

Box Class

```
public class Box implements Holder, Holdable
{
  private Weight _weight;
  private Dimensions _size;
  private HoldableBag _bag;
  public Box(Weight emptyWeight, Dimensions size)
  {
    _weight = emptyWeight;
    _size = size;
    _bag = new HoldableBag();
  }
}
```

Box class continued

```
  public void add (Holdable toBeHeld){
    _bag.add(toBeHeld);
    _weight.increaseBy(toBeHeld.getWeight());
  }
  public Holdable release(Holdable toBeReleased){
    Holdable holdable;
    holdable = _bag.release(toBeReleased);
    _weight.decreaseBy(toBeReleased.getWeight());
    return holdable;
  }
  public Weight getWeight(){
    return _weight;
  }
  public Dimensions getDimensions() {
    return _size;
  }
}
```

Box

- As currently defined box has infinite capacity. How might we correct this?
- Box might have a maximum weight.
- Dimensions might allow for the comparison of dimensions to see if it would fit. But would be complicated since the dimensions would need to know the dimensions of everything currently in the Box.

Using Interfaces

- Notice that the interface `Holdable` was used as a parameter and return type in the class `Box`.

```
public void add (Holdable toBeHeld)
public Holdable release(Holdable toBeReleased)
Holdable holdable;
```

- Interfaces can be used:
  - As formal parameter types
  - As return types
  - To declare variables
- Interfaces can not be used to instantiate instances.

How does this work for parameter passing?

Suppose a formal parameter is an interface.

- Any instance of a class that explicitly implements the interface may be passed as the actual parameter.
- Note: if a class which is a superclass explicitly implements an interface; then all of its subclasses explicitly implement the interface through inheritance.
- Only methods defined in the interface can be used by a method which has a formal parameter defined as an interface.

More about Using Interfaces

- When interfaces are used as return types and variables they state that the instance returned or declared implements the interface.
- If used for a variable, satisfying the interface is all that can be guaranteed for the variable, even though it is instantiated as some class.
- Interfaces can be used anywhere you would specify an abstract class. As with abstract classes interfaces can not be used to instantiate an instance.

Example:class Box

The following statements are all legal statements in Java.

```
Weight boxWeight;
boxWeight = new Weight (1,  ounce  );
Dimensions boxDimensions;
boxDimensions = new Dimensions(15, 20, 10,  inches  );
private Box _box;
_box = new Box(boxWeight,boxDimensions);
private Holder _holder;
_holder = new Box(boxWeight,boxDimensions);
private Holdable _holdable;
_holdable = new Box(boxWeight,boxDimensions);
```

Returning to the Box class

- The box class implements both `Holder` and `Holdable`
- Declaring a `Holder` variable (or parameter) and implementing it as a `Box`
  - Using `boxWeight` and `boxDimensions` from the previous slide
  - `Holder myStorage = new Box(boxWeight, boxDimensions);`

myStorage

- Although `myStorage` is instantiated as a `Box`, it is not valid to use the box methods `getWeight()` and `getDimensions()` with `myStorage`, since those are not methods associated with `Holder`
- `myStorage` can not be passed to any parameter which is expecting a `Box`, nor can it be assigned to a variable which is declared to be a `Box`
- The declared value of the variable `myStorage` determines how it can be used!
- The methods for add and release defined for the `Box` class will determine how `myStorage` performs those messages.

Extending Interfaces

- Suppose I want an interface which is a superset of another interface
- For example: A professional mover is a mover

```
public interface ProfessionalMover extends Mover
{       public abstract Currency chargeFee();
}
```

- In this case to implement ProfessionalMover a class would need to implement all Mover methods and the method chargeFee()
- Interfaces can only extend other interfaces!

Can we combine interfaces to make another interface?

- Suppose we want a class which must implement both Holder and a Mover.

```
public interface Transporter extends Holder, Mover
{       public abstract Currency chargeFee();
}
```

- The Transporter interface requires that all methods of `Holder` and all methods of `Mover` interfaces be implemented in addition to the method `chargeFee()`

Why build interfaces which are combinations of interfaces?

- We may have methods which use methods of more than one interface.
- A method can only use the methods available to its declared class and to the declared class or interface of its parameters. Therefore, we would need an interface definition which contains all of the methods needed as the declared type for the parameters.

Mouse Events

The MouseEvents includes all events where the mouse is:

- Pressed
- Released
- Clicked
- Entered
- Exited

The MouseMotionEvents includes all events where the mouse is:

- Dragged
- Moved

Listeners for the mouse

- MouseListener is an **interface** for MouseEvents.
- MouseMotionListener is an **interface** for the MouseMotionEvents.
- MouseAdapter is a **class** which implements both MouseListener and MouseMotionListener with *stub* methods for all seven required methods.

An Example

A program which will change the color of the shape through the cycle of colors red→blue→green→yellow. The panel will have two objects: an ellipse and a rectangle, where each will cycle independently.

Class MyApp

```
import java.awt.*;
import javax.swing.*;
public class MyApp extends JFrame
{   public MyApp(String title)
    {  super(title);
        this.setSize(1000, 800);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
        // create and add Panels to the Frame
        this.add(new ObjectPanel());
        this.setVisible(true);
    }
    public static void main()
    {       MyApp app = new MyApp("Mouse␣Example");
    }
}
```

Class ChangeEllipse

```
import java.awt.*;
public class ChangeEllipse extends SmartEllipse
{   public ChangeEllipse()
  {   super(Color.RED);
  }
  public void changeColor()
  {  Color currentColor = getFillColor();
    if (currentColor.getRGB() == Color.RED.getRGB())
    super.setFillColor(Color.BLUE);
    else if (currentColor.getRGB() == Color.BLUE.getRGB())
      super.setFillColor(Color.GREEN);
    else if (currentColor.getRGB() == Color.GREEN.getRGB())
      super.setFillColor(Color.YELLOW);
    else super.setFillColor(Color.RED);
    }
}
```

Class ObjectPanel

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;
public class ObjectPanel extends JPanel
{  private ChangeEllipse _ball;
  private ChangeRectangle _rect;
  public ObjectPanel()
    super();
     // set up the display on the panel
    this.setBackground(Color.BLACK);
    _ball = new ChangeEllipse();
    _ball.setSize(100,100);
    _ball.setLocation(100, 100);
     _rect = new ChangeRectangle();
    _rect.setSize(150, 100);
     _rect.setLocation(350,250);
    // add the panel as a MouseListener
    MyMouseListener myMouseListener = new MyMouseListener(this);
    this.addMouseListener(myMouseListener);
  }
```

Class Object Panel

```
    public void paintComponent(Graphics aBrush)
    {   super.paintComponent(aBrush);
        Graphics2D aBetterBrush = (Graphics2D)aBrush;
        _ball.fill(aBetterBrush);
        _rect.fill(aBetterBrush);
    }
    public void mouseClicked(MouseEvent e)
     {  Point currentPoint;
         currentPoint = e.getPoint();
         if(_ball.contains(currentPoint))
             _ball.changeColor();
         else if(_rect.contains(currentPoint))
             _rect.changeColor();
         _myPanel.repaint();
        }
    }
}
```

Class ObjectPanel

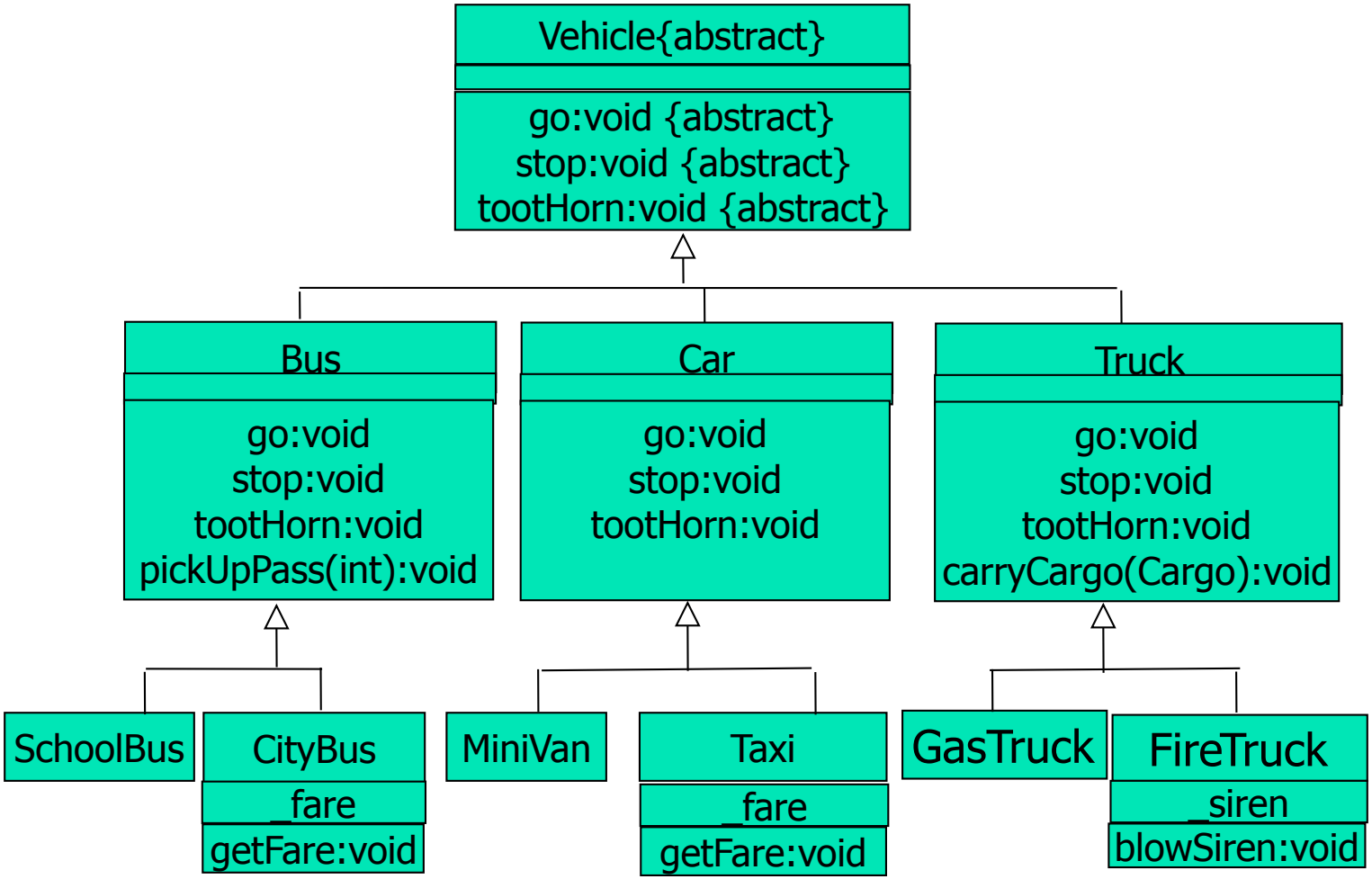
```
public class MyMouseListener extends MouseInputAdapter
{
        ObjectPanel _myPanel;
    // constructor
    public MyMouseListener(ObjectPanel bp)
    {
            _myPanel = bp;
    }
        public void mouseClicked(MouseEvent e)
        {
            _myPanel.mouseClicked(e);
    }
}
```

Polymorphism

With Inheritance and Polymorhism

- The patrons of the restaurant do not need to know who a `FoodPrepPro` is or how they prepare food. By issuing the command they know that food will be prepared and returned to them. Each `FoodPrepPro` will prepare food in their own way.

Polymorphism



- Suppose we also have a traffic cop, who is to issue tickets to vehicles.
- The type of the formal parameter, must match the type of the actual argument.
- Therefore, without polymorphism, we would need to have a method for each type of vehicle.

Relaxed Parameter Matching

- With polymorphism and inheritance we can relax the restriction on parameter matching, allowing a traffic cop to issue tickets to all types of vehicles.
- Actual parameter must be an instance of the formal parameter class or an instance of a subclass of the formal parameter.

More Generally:

- A variable declared to be a certain type can be set equal to any instance of its subclasses.

```
Vehicle _nellyBelle;
```

- The following are all valid.

```
_nellyBelle = new MiniVan();
_nellyBelle = new FireTruck();
_nellyBelle = new Bus();
```

Returning to the Traffic Cop Problem

- The traffic cop can have a method

```
public void writeTicket(Vehicle);
```

- Since the `writeTicket` method is expecting a `Vehicle` as its parameter, it can only use those instance variables and methods which are known within class `Vehicle`.
- However its actual parameter can be any subclass of `Vehicle`, since it inherits all of the instance variables and methods of `Vehicle`.

Invoking the writeTicket Method

- Given the following declarations and the vehicle class described in the diagram:

```
TrafficCop roger;
MiniVan _loveBug2;
Bus _blueLineBus;
Vehicle _nellyBelle;
```

- The following are valid messages.

```
roger.writeTicket(_loveBug2);
roger.writeTicket(_blueLineBus);
roger.writeTicket(_nellyBelle);
```

Implementing writeTicket

```
public class TrafficCop
{ . . .
  public void writeTicket(Vehicle perpetrator)
  {
    perpetrator.stop();

    perpetrator.go();
  }
}
```

- All methods invoked on perpetrator must be defined for instances of `Vehicle`.

How _nellyBelle Responds to Messages

- As long as `Vehicle` has a `go` method, which is overridden by its subclasses, `_nellyBelle.go();` will act differently based on how `_nellyBelle` is currently instantiated.
- When `_nellyBelle` is instantiated as a `FireTruck`. `_nellyBelle` will `soundSiren`, then `goQuickly`. When `_nellyBelle` is instantiated as a `CityBus` `_nellyBelle` will `giveOutAPuffOfSmoke`, `goSlowly`.

How Does This Work?

- `Vehicle` would have to have a method `go`.
- This method might be `abstract`, forcing all subclasses to have their own version of `go`, but does not have to be.
- Polymorphism allows us to build methods to implement `go` which all have the same signature.
- `_nellyBelle` responds based on the actual type of the instantiated object.

Dynamic Binding

- When method resolution is done at run time, it is called **dynamic binding**.
- In the `writeTicket` method invocation perpetrator is bound to an actual argument. The specific class of that argument is not know until `writeTicket` is invoked. Therefore the particular implementation of go is not known until run time.
- Hence the need for dynamic binding.

More about Polymorphism

- Since `_nellyBelle` is declared as `Vehicle` (even though it may be instantiated as a subclass of `Vehicle`) it may only be sent messages appropriate to `Vehicles`.
- `_nellyBelle.tootHorn();` or
- `_nellyBelle.go();` may be appropriate, while
- `_nellyBelle.blowSiren();` is not appropriate since blowSiren is not defined in the `Vehicle` class.

Resolving Methods with Polymorphism

- Look at the actual type of the instance (at run time).
- Look for a method in that class with the appropriate signature.
- If none is found move up the hierarchy, until a method with the appropriate signature is found.

Why is Dynamic Binding Needed?

- If you send a go message to a `Vehicle` although you may not know what type of Vehicle it is you would expect it to "go", in a manner appropriate for its subclass.
- Dynamic binding allows this.

Summary of Polymorphism with Inheritance

- There is a class hierarchy.
- There are outsiders sending messages to some object in the hierarchy.
- The outsiders may not know, nor do they need to know, the position of things in the hierarchy.
- Objects may respond in different ways depending on their position in the hierarchy.

What makes Polymorphism work?

- The actual type of a variable can be the same as the declared type (except for abstract classes), or any subclass of the declared type.
- The messages that can be sent to an object depend on its declared type.
- The way an object responds depends on its actual type.
- Java has dynamic binding: that is, given a number of methods with the same name, it doesn't decide until run time which method will be executed.

Static Binding

- In non-object oriented languages, or when polymorphism is not an issue:
  - Binding is based on the formal parameter(s) and can be done at compilation time.
- This is called **static binding**.

Polymorphism Tradeoffs

- Declared type of variable determines which methods can be sent to any instance. only methods defined in the superclass work
- if subclass defines additional methods, they are not available, in code written based on superclass.
- But, if code works for declared class, it will work for any subclass without changes.

Partial overriding of methods

- We want to reuse the code of the superclass, while adding additional functionality to it, rather than totally replacing it.
- How do we do it?
  - call superclasses method of same name using

    ```
    super.<method_name>();
    ```

  - plus provide additional code.

super.

- Can choose to access superclass's public methods instead of my own anywhere by using

  `super.<methodName>()`

- Can call method of superclass multiple times if you want, by invoking it more than once.

Review

- `super.<methodName>()`
  - Invokes superclass's method, i.e. start method resolution one level up in hierarchy.
- `super(<parameters>)`
  - Invokes superclass's constructor.
  - Generally used in constructor.
  - Parameters optional, if not used invokes superclass's default constructor.

More Review

- `this.<methodName>()`
  - Used to invoke another method of the same class with the object passed the message.
  - Can be written more simply as: `<methodName>()`
- `this(<parameters>)`
  - invokes a constructor of the same class as it, with different parameters
- `this`
  - refers to the instance of the class which was passed the message.
  - Used when this object must be passed to another method or returned.

Polymorphism can be Tricky!

- Suppose we have a class Garage which has the method

```
public void storeCar(Sedan myCar){    }
```

- And, we construct a local variable with declared type Car and actual type Sedan.

```
Car niceCar = new Sedan(int);
```

- Can we pass niceCar as a parameter to storeCar?

```
Garage _garage=new Garage();
_garage.storeCar(niceCar);
```

No! Why?

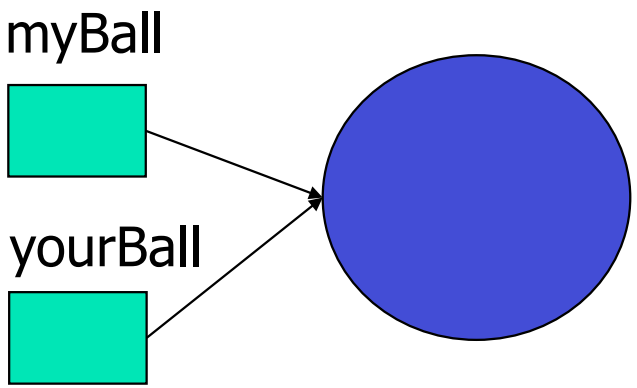
- The declared type of the actual parameter must be the same class or a subclass of the formal parameter!!
- Since the declared type of `niceCar` is `Car`, a superclass of `Sedan`, not a subclass of `Sedan`, it can not be passed as a parameter to `storeCar`.
- `storeCar` expects a Sedan, and uses the methods of a Sedan, while `niceCar` only supports the methods of `Car`.

A Word About Equality

- **Identity equality** - both point to the same object or for primitives are the same value.
- **Property equality** - all instance variables have the same values, i.e. if they are references they point to the same objects.
- **Deep Property equality** - all primitive types are the same, when property equality is used on the objects components and peers all the way down to the primitive types.

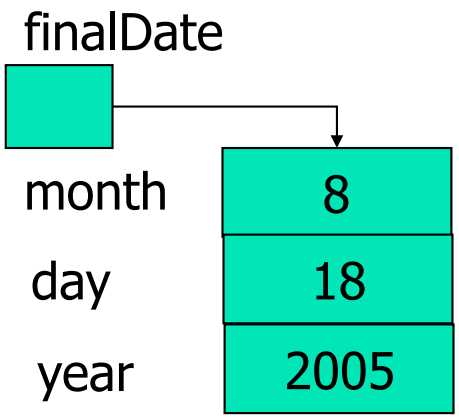
Identity Equality

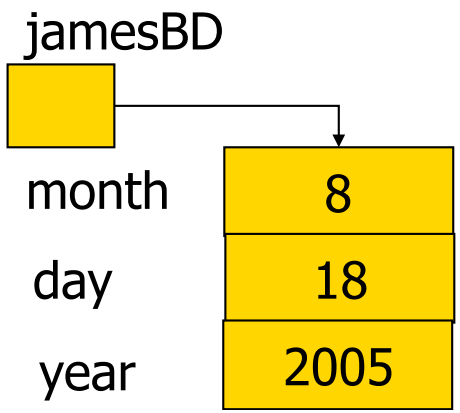
```
Ball   myBall, yourBall;
myBall = new Ball();
yourBall = myBall;
```



Property Equality

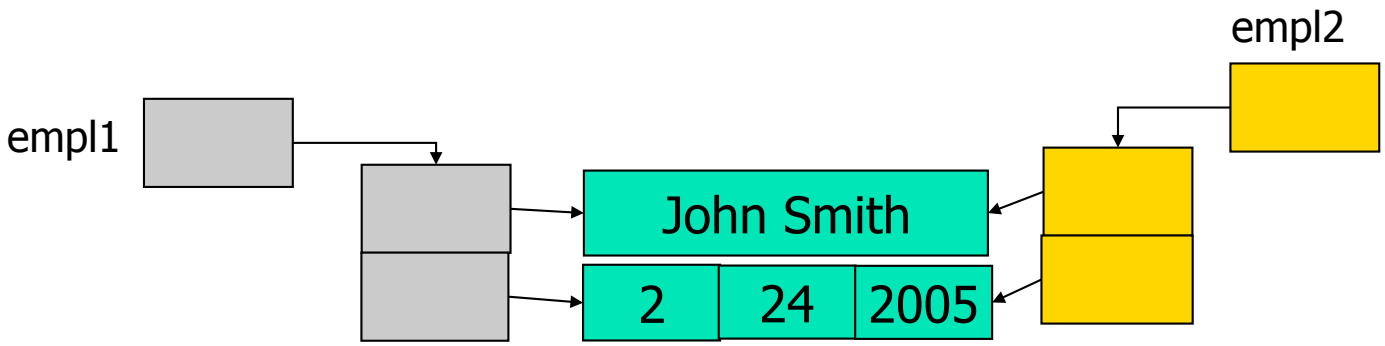
```
Date finalDate = new Date(8,18,2005);
Date jamesBD = new Date(8,18,2005);
finalDate == jamesBD
```





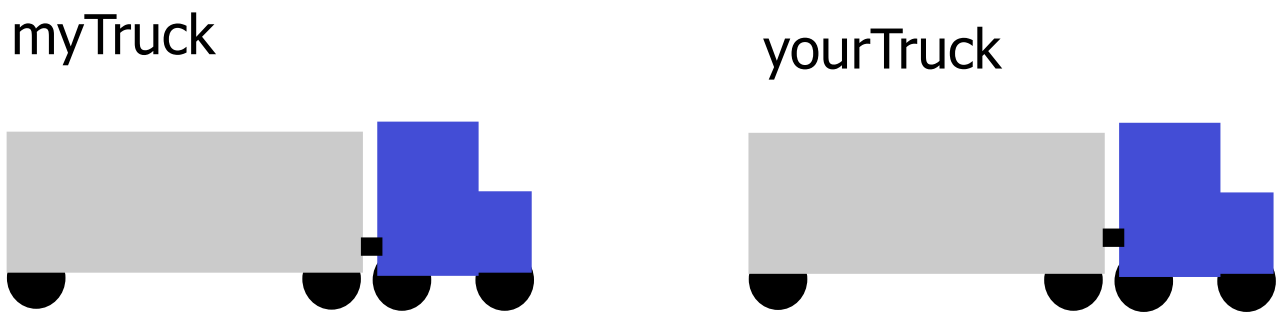
Another Example

```
String name = new String( John Smith );
Date hireDate = new hireDate(2, 24, 2005);
Employee empl1, empl2;
empl1 = new Employee(name, hireDate);
empl2 =new Employee(name,hireDate);
```



Deep Property Equality

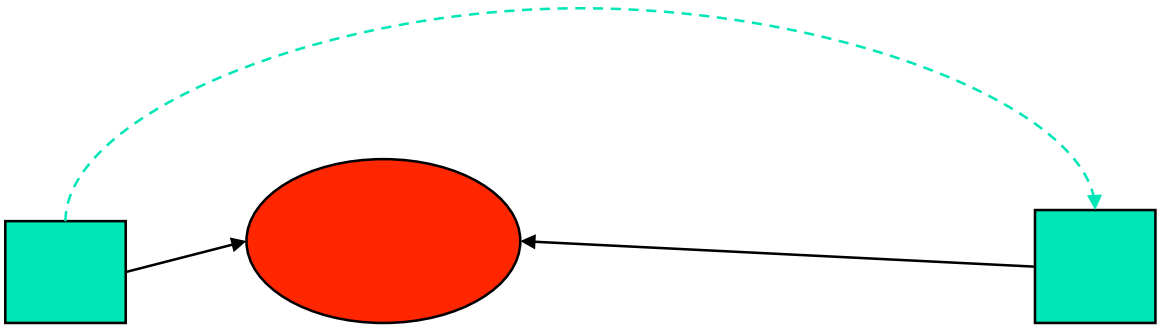
```
import java.awt.Color.*;
Truck myTruck, yourTruck;
myTruck = new Truck(new Cab(Color.BLUE),new Cargo(Color.GRAY));
yourTruck = new Truck(new Cab(Color.BLUE),new Cargo(Color.GRAY));
```



What Does it Mean?

- The == and != operators always perform Identity Equality.
- If you want PropertyEquality or Deep PropertyEquality you must write a method for the class with the name equals

Parameter Passing - Objects

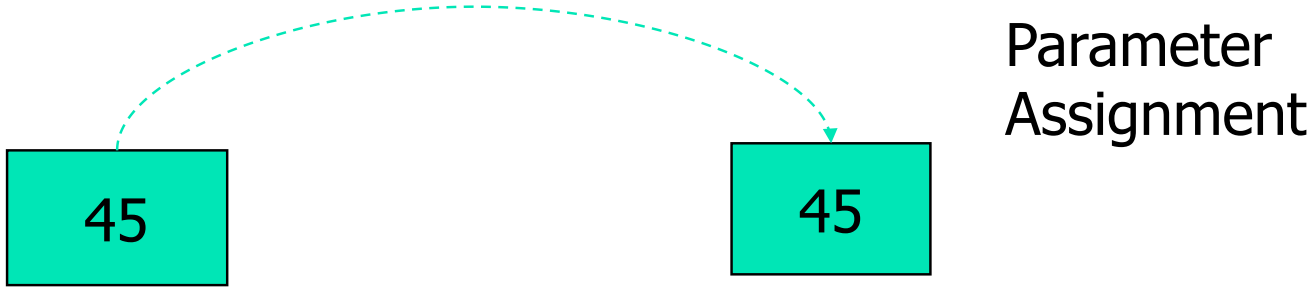


Actual parameter

Formal Parameter

i.e. Pass by reference

Parameter Passing - Primitive Types



Actual parameter

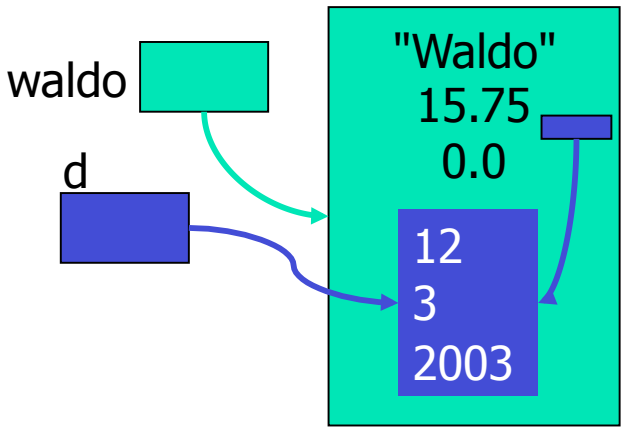
Formal Parameter

i.e. Pass by value

Another look at Accessor Methods

- The value returned, when returning an object is a reference to the object!
- Problem:

```
Employee waldo;
waldo = new Employee( Waldo",15.75,0.0,new Date(2,1,2004));
Date d;
d = waldo.getHireDate();
```

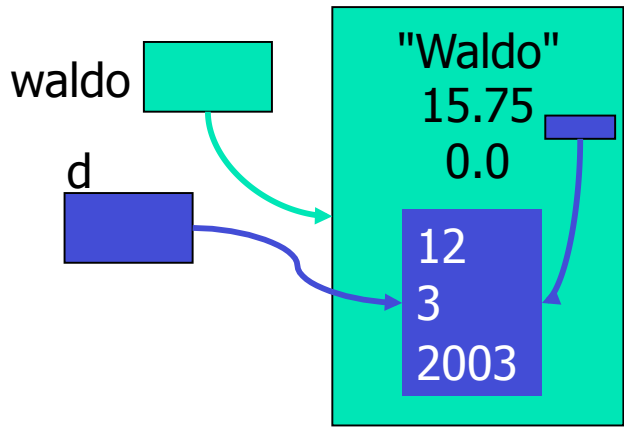


Another look at Accessor Methods

- The value returned, when returning an object is a reference to the object!
- Problem:

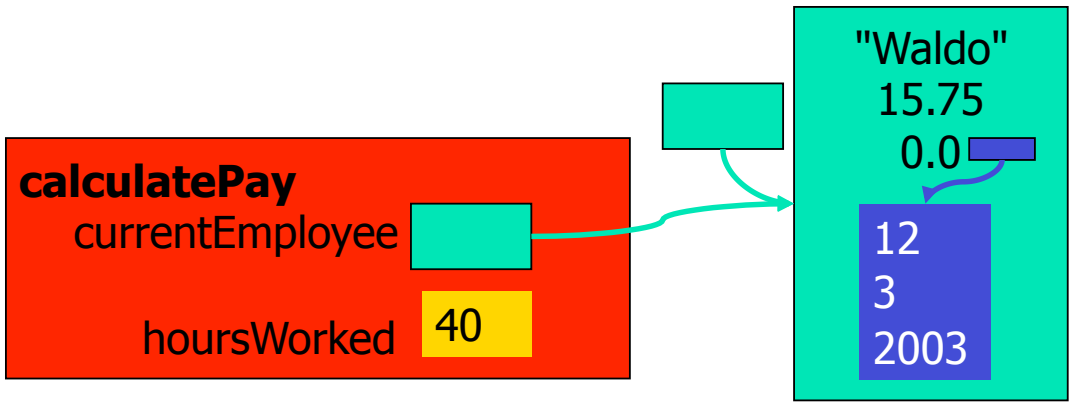
```
Employee waldo;
waldo = new Employee( "Waldo",15.75,0.0,new Date(2,1,2004));
Date d;
d = waldo.getHireDate();
d.setDate(12,3,2003);
```

Even though Employee may not have a Mutator for HireDate the client directly modifies HireDate.



Another Example

- Although you might think of parameter passing as being call by value, this doesn't help when we pass in a copy of a reference to an object.
- `calculatePay(waldo, 40);`
- The `calculatePay` function can modify
- `waldo` using its mutator methods.
- `currentEmployee.setTotalPay();`



Shallow Copy vs. Deep Copy

- A **shallow copy** copies only the class data members, and does not copy any pointed-to data.
- A **deep copy** copies not only the class data members, but also makes separately stored copies of any pointed-to data.

Whats the difference?

- A **shallow copy** shares the referenced data with the original class object.
- A **deep copy** stores its own copy of the referenced data at different locations than the data in the original class object.

Clone Method

- Clone is a method of the Object class
- Object is the parent class of all classes.
- Creates a shallow copy of an object

```
public Date getHireDate() {
  return (Date) _hireDate.clone();
}
```

Clone Method

- **Accessor methods** - Clone will help some in the case of a return value. Provided the object consists only of primitive data types and constants.
- **Mutator methods** and other methods where you want call by value for parameter passing - Clone method helpful here with the same caveat.

Wrapper Classes

- Java provides class types that work similar to the primitive types, for when objects are needed.
- The object is immutable however.

Integer Object Example

- Creating an Integer object

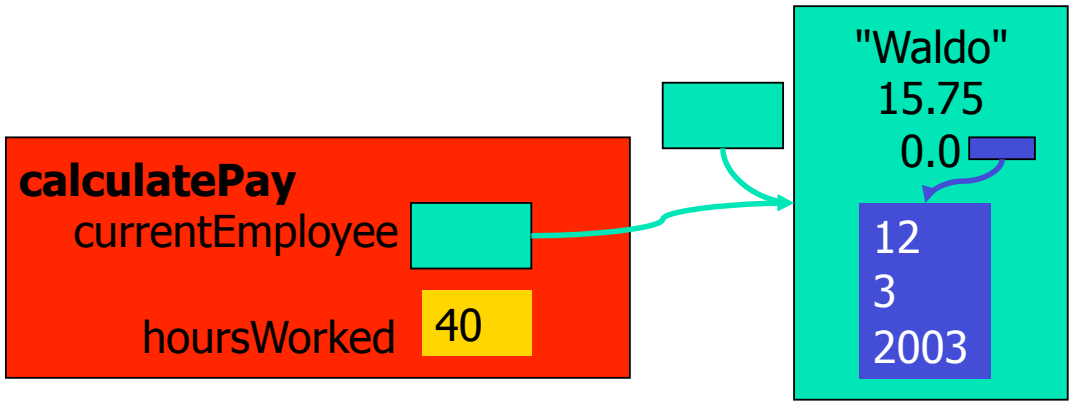
```
Integer integerObject = new Integer(15);
```

- Unwrapping or unboxing the object

```
int i = integerObject.intValue();
```

- There is no method to set the value of an Integer other than the constructor

Wrapper Classes



Wrapper Classes

- Have no default constructor.
- Provide many useful methods
- Provide useful constants such as
- `Integer.MAX_VALUE` and
- `Integer.MIN_VALUE`

Passing Primitive Types by Reference

- To modify, what would be a primitive type, you must create a class to hold the primitive and has both accessor and mutator methods.
- Then using the accessor and mutator methods you can manipulate the primitive data inside the object.
- Return the object and then access its data outside the method.

Parts of Java API We Will Be Using Extensively

- **Graphics2D** - code that lets us draw shapes and text on the screen and manipulate shapes and text
- **Swing** - collection of tools for windows, panels to draw on, and user interface objects like buttons menus, text boxes, and scroll bars
- **Events** - classes/interfaces that allow us to deal with events generated by Swing objects and user interactions

AWT and Swing

- AWT: Abstract Windowing Toolkit
  - classes for windows and other graphical components "old" around since Java 1.1
- Swing:
  - same basic stuff as AWT
  - all "lightweight" components
  - written in Java so they are platform independent
  - allows more control over "look and feel" of application
  - cleaner design
  - introduced in Java1.2 which contained both Swing library and Java2D classes in AWT library
  - built on top of AWT making substantial improvements

Frames

JFrame in Swing (`javax.swing.JFrame`)

- Provides window to display the program

Window has:

- a border
- a title
- min/max and close buttons

Recipe for building graphics programs

```
public class BallApp extends javax.swing.JFrame {
  public BallApp (String title) {
    super(title);
    this.setSize(600, 450);
    this.setDefaultCloseOperation(
    javax.swing.JFrame.EXIT_ON_CLOSE);
    // add code for panels here
    this.setVisible(true);
        }
  public static void main (String [ ] args) {
    BallApp app = new BallApp ("My First JFrame");
  }
}
```

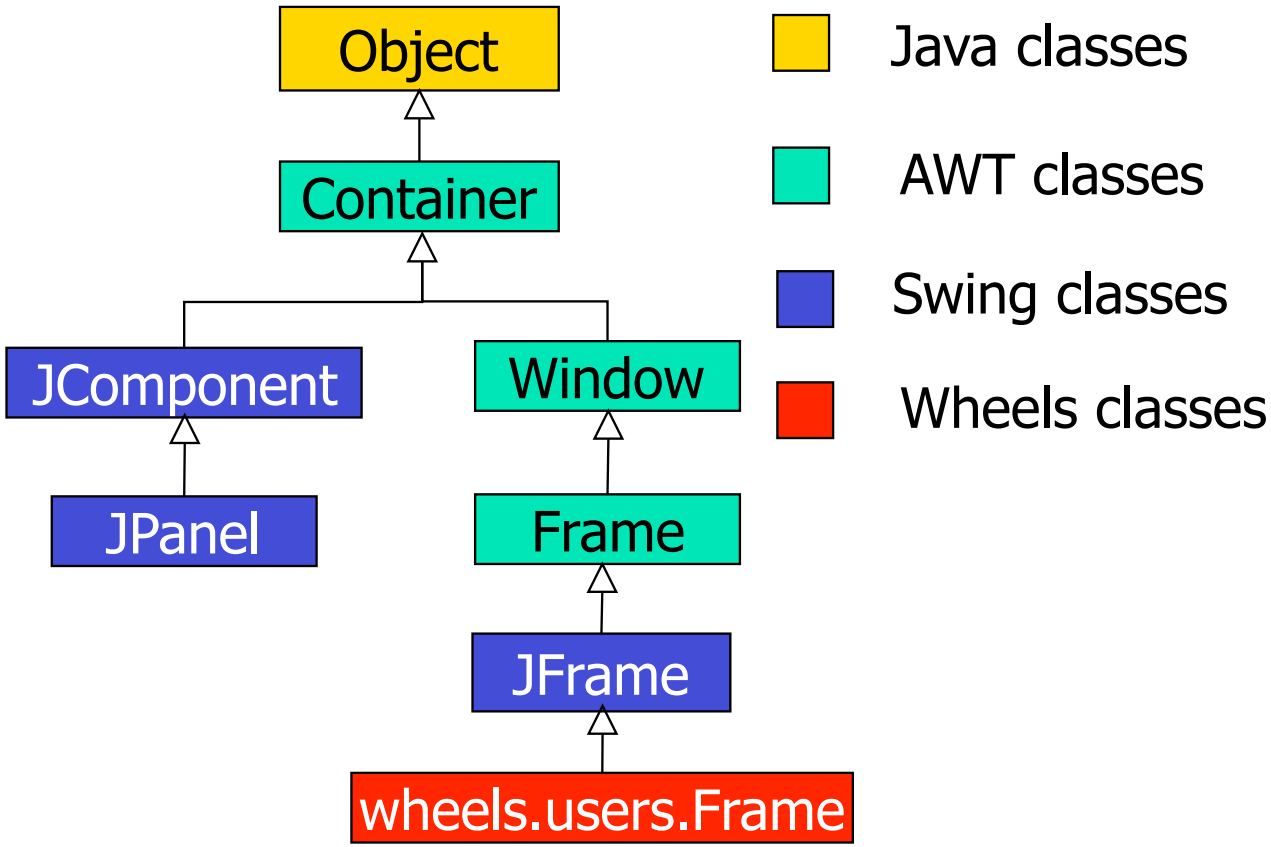
Questions on Previous Slide

- Can the title be changed on a `JFrame`?
  Yes. java.awt.Frame class of which `JFrame` is a subclass has a `setTitle` method.
- Can the window be set to full screen width?
  Yes but this is more complicated. You can make the window resizable by the user, which should allow the user to make it full screen size.

JPanel

- `JPanels` (and their subclasses) hold graphical objects.
- `JPanels` are placed inside of a `JFrame`.
- They are like the canvas, on which you might paint a picture.

Hierarchy of JFrames and JPanels



Drawing in Swing

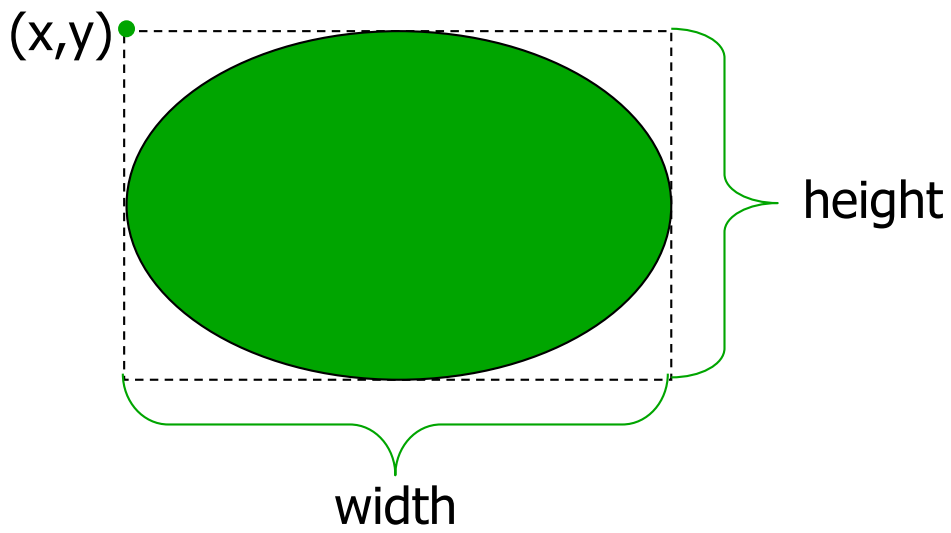
- I can draw something to a panel, but it will generally not appear until the panel is "painted" on the screen.
- This is useful if a window is under another window.
- Updating is not automatic
- When an object is modified (moved or changed color or size) it will need to be `repainted`.

Things You Need to Know

- repaint method
  - `repaint()` schedules a panel to be updated
  - `repaint(int x, int y, int width, int height)` schedules panel within the specified bounds to be updated
  - `repaint(Rectangle r)`
- `paintComponent` method
  - invoked when repaint is called
  - invoked when `JPanel` is opened, resized, or another window in front of the `JPanel` is closed
- Graphics object
  - parameter to paintComponent
  - a collection of attributes, not a shape

Bounding Box

The bounding box is the rectangle which circumscribes the Shape.



An example: setColor()

```
public void setColor(java.awt.Color c)
{
        _c = c
        _dp.repaint(this.getbounds());
}
```

- _c is an instance variable of type java.awt.Color
- _dp is an instance variable for the Drawing Panel which contains the object
- getbounds() method to get the Shapes bounds

Java Colors

13 built-in color constants

`java.awt.Color.RED`
`java.awt.Color.BLUE`
`java.awt.Color.GREEN`
`java.awt.Color.YELLOW`
`java.awt.Color.MAGENTA`
`java.awt.Color.ORANGE`
`java.awt.Color.PINK`
`java.awt.Color.CYAN`
`java.awt.Color.BLACK`
`java.awt.Color.DARK_GRAY`
`java.awt.Color.GRAY`
`java.awt.Color.LIGHT_GRAY`
`java.awt.Color.WHITE`

- Other colors can be created using the `Color` constructor which takes three `int` parameters, (red, green, blue), where each has a value between 0 and 255

paintComponent

- You must override paintComponent in your JPanel if you want to display more than just the panel. For example, override paintComponent to display your graphical Shapes.

```
public void paintComponent (Graphics g)
{
  super.paintComponent(g);
  // details about how to draw the contents of JPanel
}
```

Using a JPanel

To set up a JPanel and draw in it

1. Make a class that extends `JPanel`
2. Include your own `paintComponent`
3. Invoke `repaint()` when it needs to be updated

Class BallPanel

```
public class BallPanel extends javax.swing.JPanel{
        public BallPanel () {
                super();
                this.setBackground(java.awt.Color.BLUE);
        }
        // called by the system when the panel needs repainting
        public void paintComponent (java.awt.Graphics aBrush){
                super.paintComponent(aBrush);
                // insert code to paint contents of panel
        }
}
```

Adding a JPanel to the JFrame

- Replace the comment in the JFrame recipe with:
- `this.add(new BallPanel());`

Graphics and Graphics2D

- While Graphics can draw objects using methods all the methods have different names
- Graphics2D extends Graphics

`drawOval(x,y,w,h)` `drawRect(x,y,w,h)`

`drawLine(x,y,x2,y2)` `drawEllipse(x,y,w,h)`

  - Graphics2D has polymorphic draw and fill methods that work on Shapes
  - Graphics2D supports more general manipulation such as translation and rotation

Graphics and Graphics2D

- The formal parameter type of the argument for `paintComponent` is Graphics
- The actual parameter type of the argument for `paintComponent` is Graphics2D
  - Therefore if pen is a Graphics2D object
  - `pen.draw(thing)` will invoke `pen.drawOval(x,y,w,h)` when thing is an `Ellipse`
  - `pen.draw(thing)` will invoke `pen.drawRect(x,y,w,h)` when thing is a `Rectangle`

SmartEllipse class

```
public class SmartEllipse extends java.awt.geom.Ellipse2D.Double{
  private java.awt.Color _borderColor, _fillColor;
  private int _rotation;
  private final int STROKE_WIDTH = 2;
  public SmartEllipse(java.awt.Color aColor){
  _borderColor = aColor;
  _fillColor = aColor;
  _rotation = 0;
  }
  // methods not provided by Java
  public void setBorderColor(java.awt.Color aColor){
    _borderColor = aColor;
  }
  public void setFillColor(java.awt.Color aColor){
    _fillColor = aColor;
  }
  public void setRotation(int aRotation){
    _rotation = aRotation;
  }
```

SmartEllipse class

```
// drawing methods not provided by Java
  public void fill(java.awt.Graphics2D aBetterBrush){
    ava.awt.Color oldColor = aBetterBrush.getColor();
    aBetterBrush.setColor(_fillColor);
    aBetterBrush.fill(this);
    aBetterBrush.setColor(oldColor);
  }
  // drawing methods not provided by Java
  public void draw(java.awt.Graphics2D aBetterBrush){
    java.awt.Color oldColor = aBetterBrush.getColor();
    aBetterBrush.setColor(_borderColor);
    ava.awt.Stroke oldStroke = aBetterBrush.getStroke();
    aBetterBrush.setStroke(new java.awt.BasicStroke(STROKE_WIDTH);
    aBetterBrush.draw(this);
    aBetterBrush.setStroke(oldStroke);
    aBetterBrush.setColor(oldColor);
  }
}
```

BallPanel class

```
public class BallPanel extends javax.swing.JPanel{
  private SmartEllipse _ball;
  private final int INIT_X = 75;
  private final int INIT_Y = 75;
  private final int BALL_WIDTH = 60;
  public BallPanel () {
    super();
    this.setBackground(java.awt.Color.BLUE);
    _ball = new SmartEllipse(java.awt.Color.RED);
    _ball.setLocation(INIT_X, INIT_Y);
    _ball.setSize(BALL_WIDTH, BALL_WIDTH);
  }
  // called by the system when the panel needs repainting
  public void paintComponent (java.awt.Graphics aBrush){
    super.paintComponent(aBrush);
    java.awt.Graphics2D betterBrush = (java.awt.Graphics2D) aBrush
    _ball.fill(betterBrush);
  }
}
```

Animation and Events

- Building an animation application.
- The objects in the animation seem to move around with no user input.
- Uses a Timer.
- Timer generates a message to other objects in the program at fixed intervals, and the program moves the objects a certain amount at each "tick"
- The Timer is an event, like the mouse events.

How Events Work

- Some object generates or detects events
  - mouse click, mouse press, mouse release, mouse drag
  - key press
  - button press, window close
- Objects that are "interested" register with the generator as a `Listener`
- When generation object produces/detects an event it sends a message to each `Listener`
- The `Listener` object then process the event by overriding `actionPerformed(ActionEvent e)` method

More Details

- A timer generates periodic events of class ActionEvent
- If some class wants to be notified of Timer events
  - `import java.awt.event.*`
  - implement the ActionListener interface
    - which has one method
      `void actionPerformed(ActionEvent e)`
- register with the Timer instance, if `myTimer` is a Timer instance, place the following statement in the class definition of the Listening class
  `myTimer.addActionListener(this);`

Built in Shapes in AWT

- `java.awt.geom.Ellipse.Double`
- `java.awt.geom.Line2D.Double`
- `java.awt.geom.Arc2D.Double`
- `java.awt.geom.Rectangle`
- `java.awt.geom.Rectangle2D.Ellipse`

Example

Putting it all together

- Write a panel with an ellipse which changes color from red -¿ green -¿ red every tick of the timer.

FlashBall Class

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.geom.*;
public class FlashBall extends JPanel implements ActionListener
{   private Color _myColor;
        private Ellipse2D.Double _myBall;
        private Timer _myTimer;
        public FlashBall(double x, double y, double diameter)
        {       setBackground(Color.WHITE);
                _myBall = new Ellipse2D.Double();
                _myBall.setLocation(x,y);
                _myBall.setSize(diameter, diameter);
                _myColor = Color.RED;
                _myTimer = new Timer(1000, this);
                _myTimer.start();
        }
```

FlashBall Class

```
        public void paintComponent(Graphics g)
        {       super.paintComponent(g);
                Graphics2D pen = (Graphics2D) g;
                pen.setColor(_myColor);
                pen.fill(_myBall);
        }
        public void actionPerformed(ActionEvent e)
        {       if (_myColor == Color.RED)
                        _myColor = Color.GREEN;
                else    _myColor = Color.RED;
                repaint();
        }
}
```

MyApp Class

```
import javax.swing.*;
public class MyApp extends JFrame {
  public MyApp(String title) {
    super(title);
    this.setSize(1000,800);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.add(new FlashBall(150.0, 100.0, 600.0));
    this.setVisible(true);
  }
  public static void main (String[] args)        {
    MyApp app = new MyApp( FlashingBall );
  }
}
```

Design Patterns

- A design pattern is a way of putting components together to solve a specific problem or set of conditions.
- Design patterns allow us to use the experience of others when solving our problems.
- For each design pattern we will specify
  - the name of the pattern,
  - the problem it is designed to solve,
  - the structure of the pattern
  - the pros and cons of using the pattern.

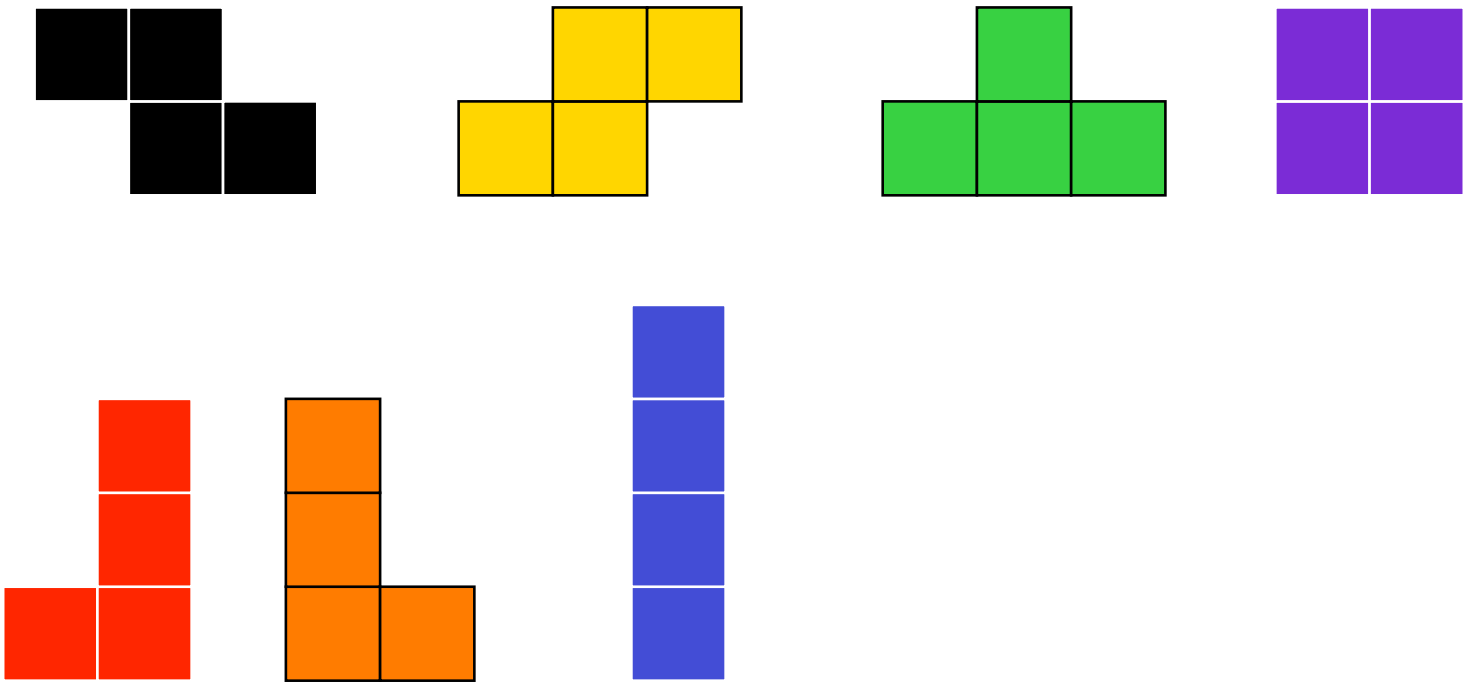
Some Design Patterns

- **Factory Pattern** - Allows for the generation of new objects of a given type, returning the new object.
- **Holder Pattern** - Allows access to a component through accessor/mutator methods.
- **Proxy Pattern** - Allows an object to send messages to another object that changes frequently.
- **Composite Pattern** - Allows sending messages to the entire object rather than to the parts of the object, such as to move a fish.

The Domain

- We will use the Tetris game in examples for some of these patterns.
- The Tetris game is composed of seven pieces which are randomly selected and move from the top of the board to the bottom of the board. While pieces are moving they can be rotated, shifted to the right or left, or simply fall.
- A piece stops falling when it either reaches the bottom of the board or it comes to rest on another piece. When an entire row is completed at the bottom of the board it is removed and all rows above it are moved down. The object of the game is to remove as many rows as possible before a piece is placed in the top row and unable to move.

The Seven Tetriminos



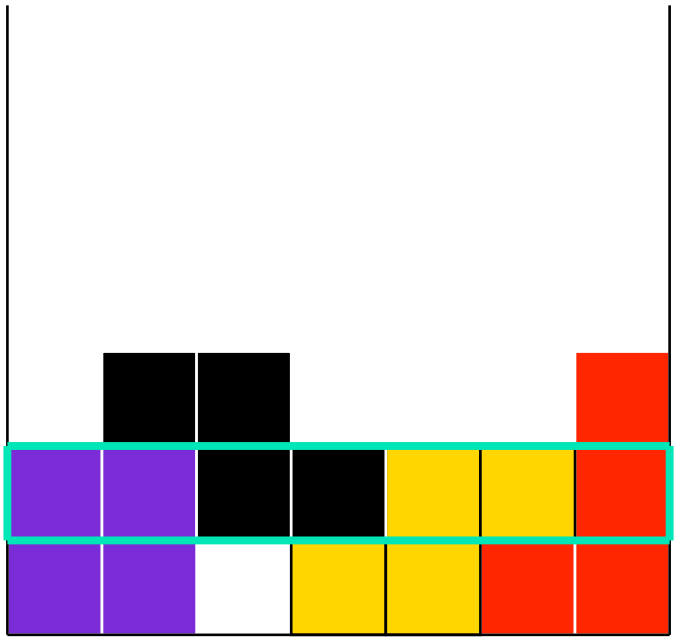
The Game

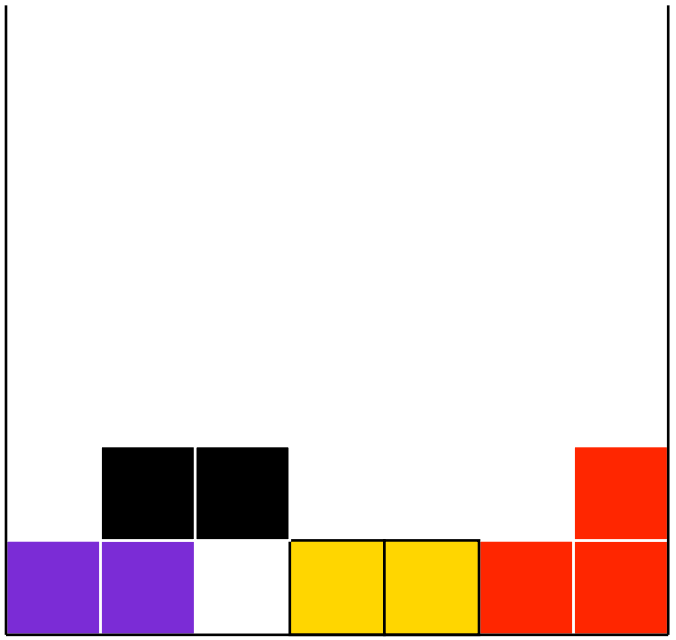
- Tetris pieces move down the board one square at a time at regular intervals.
- There is only one piece moving at any given time.
- Using the keyboard, the user can make pieces move left, right, rotate, and drop, and can pause the program.
- Two squares can not occupy the same square on the board.

More Game Details

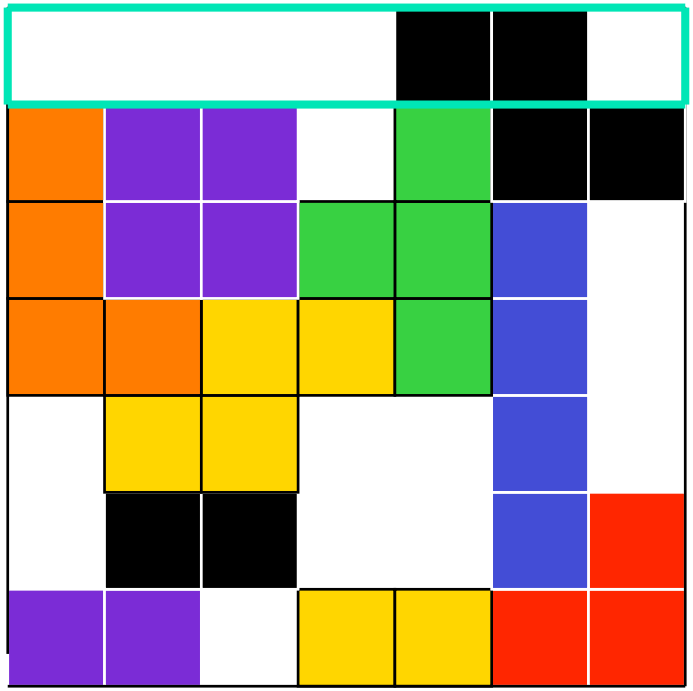
- After a piece cannot fall any further its squares become part of the board and a new piece starts falling from the top.
- When a row gets filled, it should disappear, and the rows above it should "fall" down one square.
- If the top row has a square in it then the game is over.

When a Row Disappears





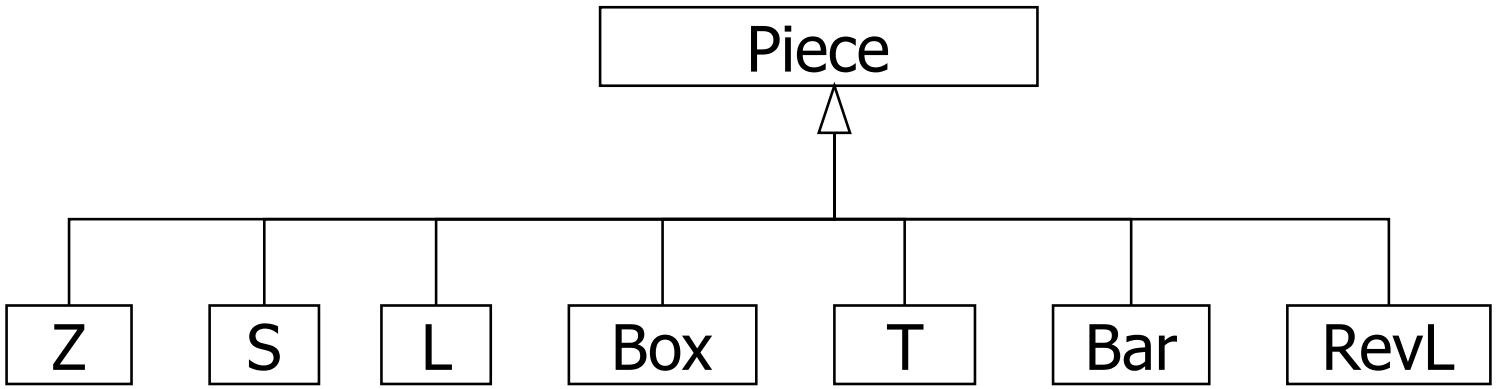
Game Over



Factory Pattern

- A factory is responsible for producing an object of a specific type and returning it.
- The factory class can produce objects of a subclass.
- In the case of Tetris we want a PieceFactory which will based on a random number to create a new piece of the appropriate type and return it.

Tetrimino Hierarchy



Piece Factory

- instance variable is of type `Piece` (superclass)
- generate random number (`java.util.Random`)
- use switch statement to decide which piece to instantiate(instantiate appropriate subclass object)
- return instance variable

Piece Factory

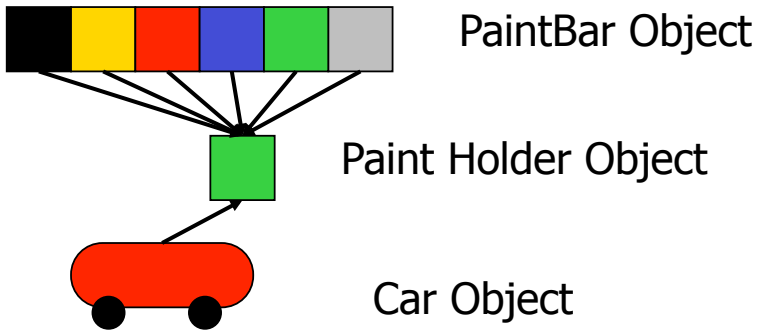
```
public Piece PieceFactory() {
  Piece newPiece;
  int randomNumber;
  randomNumber = (int) (Math.floor(Math.random()*7)+1);
  switch(randomNumber) {
    case 1: newPiece = new Z();     break;
    case 2: newPiece = new S();     break;
    case 3: newPiece = new L();     break;
    case 4: newPiece = new RevL(); break;
    case 5: newPiece = new Box();   break;
    case 6: newPiece = new Bar();   break;
    case 7: newPiece = new T();     break;
  }
  return newPiece;
}
```

Factory Pattern Recap

- **Name**: Factory Pattern
- **Problem solved**: Need to generate several different types of pieces, each a subclass of a particular class.
- **Structure**: Create a method which returns an object of the superclass. Using an instance variable of the superclass, instantiate the subclass object and return it.
- **Pros and cons**: Uses inheritance to provide a generic piece from a collection of pieces. Only through using polymorphism can the methods of the subclass be used.

Holder Pattern

- Class which holds information for other classes to work with.
- Example: The current paint color could be represented as a PaintHolder class.



Holder Pattern

- The `PaintHolder` class would have a Color instance variable (and perhaps the Rectangle), and the accessor and mutator methods.
- When the color in the `PaintBar` is changed it sets the `PaintHolder` object.
- When an object (such as the car) needs to know the current color it accesses the `PaintHolder` object.

PaintHolder Class

```
public class PaintHolder {
        private Color _currColor;
        public void setColor(Color c) {
                _currColor = c;
        }
        public Color getColor() {
                return _currColor;
        }
}
```

Holder Pattern

- Provides object that acts as holder for other object called the subject.
- acts as placeholder for subject that many other objects might reference
- holder object is "stable", instance referenced by other objects
- holder can change what subject it is referencing, including potentially instance of different subclass, without affecting those objects that reference it

Holder Pattern

- Holder object
  - contains object for which it manages changes
  - provides one level of indirection to subject object
  - provides accessor/mutator methods
- Advantages
  - easy to change object that many clients reference because those objects only refer to holder provide different interface to subject object
  - eg, subject object may be immutable, but holder provides mutable interface.
- Disadvantages
  - requires extra class and extra delegation

Proxy Pattern

- Proxy **acts on behalf of** another subject
  - proxy has reference to actual instance of subject
  - that reference can change
  - all client objects know about proxy
  - proxy has methods that match those of subject
  - clients call methods on proxy which forwards them on to the subject

Diagram of Proxy Pattern

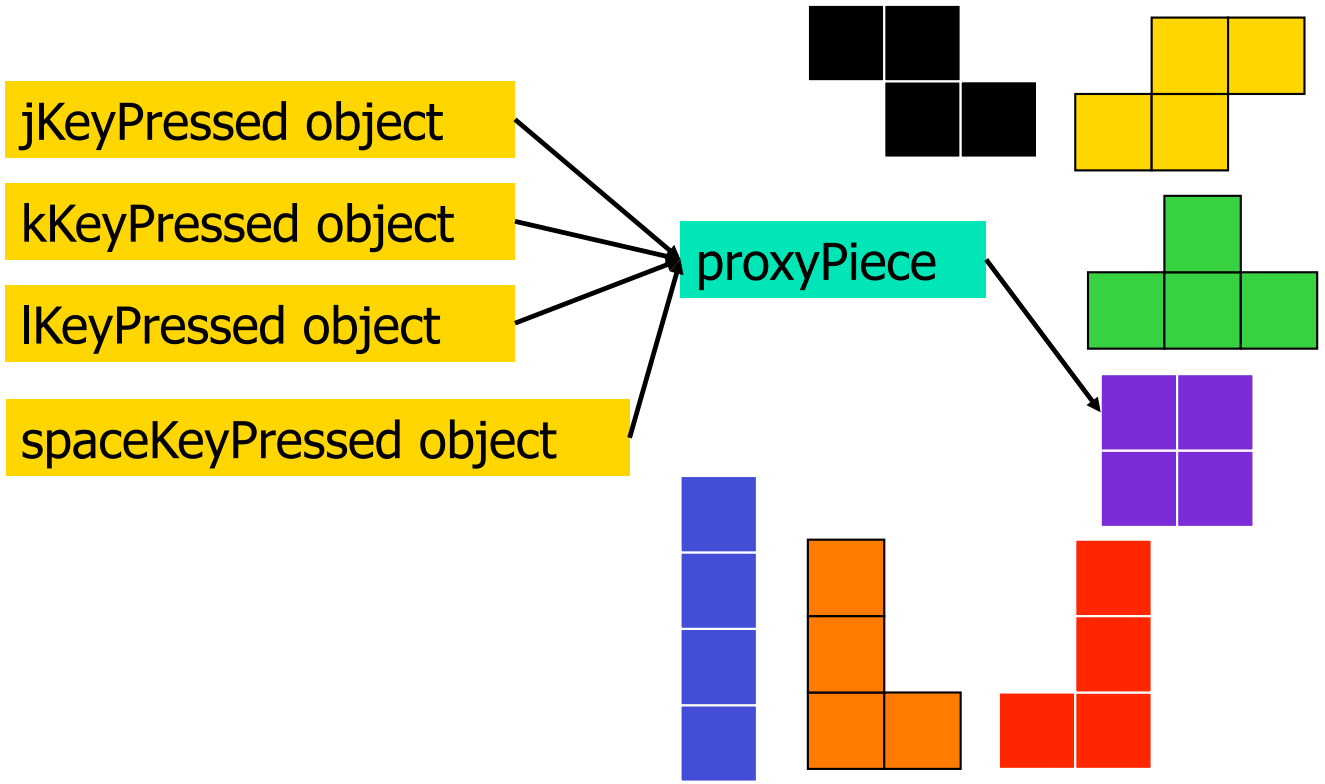
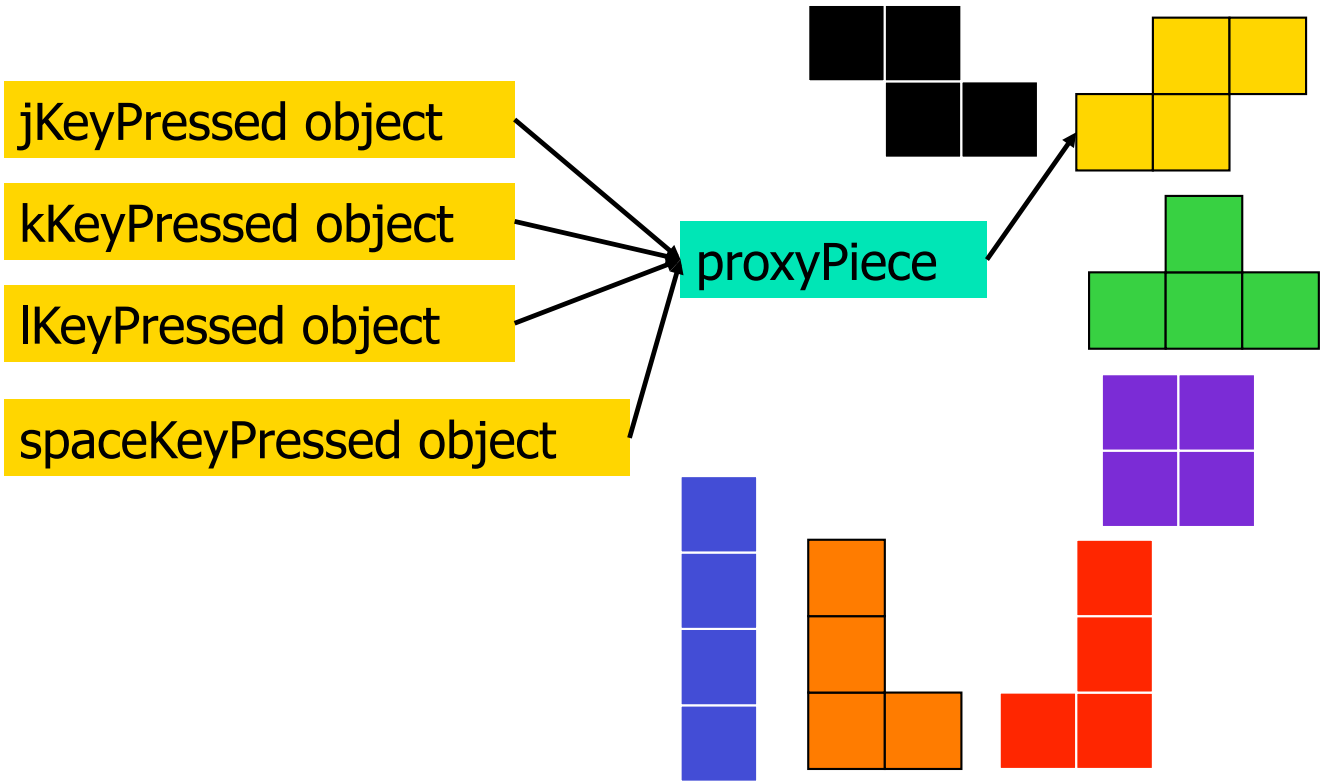


Diagram of Proxy Pattern



Proxy Interface

```
public interface PieceMove {
        public void moveLeft();
        public void moveRight();
        public void rotate();
        public void drop();
}
```

Proxy Class

```
public class ProxyClass implements PieceMove {
  private Piece _currPiece;
  // constructor does nothing
  public PieceProxy() {}
  public void setPiece(Piece piece) {
    _currPiece = piece;
  }
  public void moveLeft() { _currPiece.moveLeft();}
  public void moveRight() {_currPiece.moveRight();}
  public void rotate() {_currPiece.rotate();}
  public void drop() {_currPiece.drop();}
}
```

Proxy Recap

- **Name**: Proxy Pattern
- **Problem solved**: Ability to deal with an object, which is either expensive to create, or changes periodically. Proxies allow dealing with subclasses of objects where the particular type of the object is unknown.
- **Structure**: Create a interface which contains the methods that the proxy must be able to respond to. Create a class that implements the interface, stores an object into an instance variable, and then implements the methods by passing the message to the instance variable. One or more actor classes that implement the interface, objects of the actor classes will be assigned to the instance variable in the proxy class.

Proxy Recap

- **Pros and cons**: Using the proxy class makes the client objects simpler. Client objects like the keyboard keys dont need to know which object they are really dealing with. On the other hand we have an additional class, and object, and an additional method call (the client calls the proxy, and the proxy calls the actor, rather than the client calling the actor directly).

Holder vs. Proxy

- Similarity between Proxy and Holder patterns
  - both provide consistent interface to user while allowing the subject to change.
  - both manage the change of the subject
- Differences
  - Holder contains subject that clients can access for their use.
  - Proxy knows about subject that clients can call methods on, by way of proxy. Proxy passes those messages on to the subject.

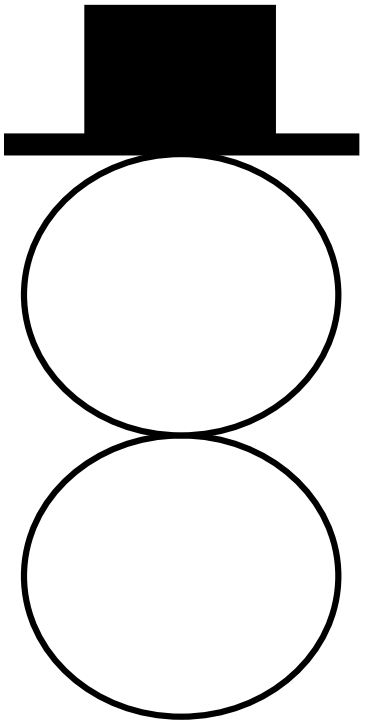
Composite Pattern

The composite pattern is used when you want to do something to and entire composite object, rather than having the client send the message to all of the components the client simply sends the message to the composite object and the composite object is responsible for dispensing the message to each component.

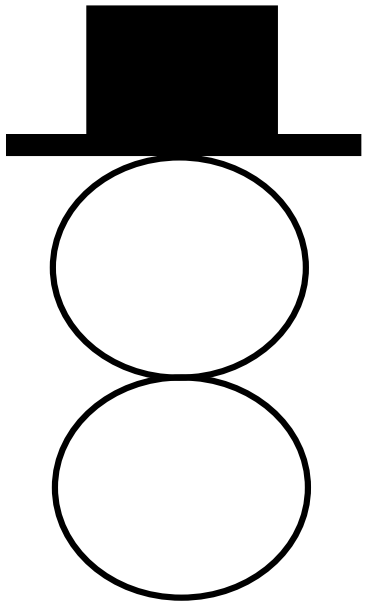
For example:

- Suppose that we wanted a snowman to melt.
- It is difficult for the client to make the changes necessary to the component parts because their size and positions are interrelated. However the client could tell the entire snowman to melt and let it take care of changing its parts.

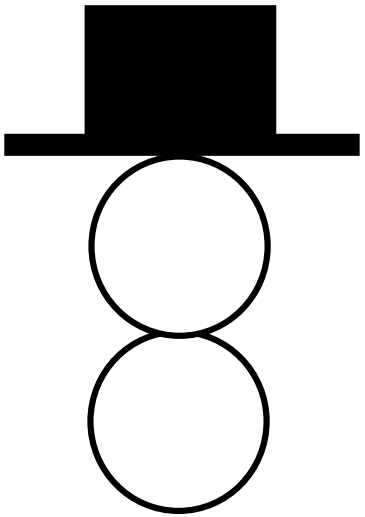
The Snowman Melting



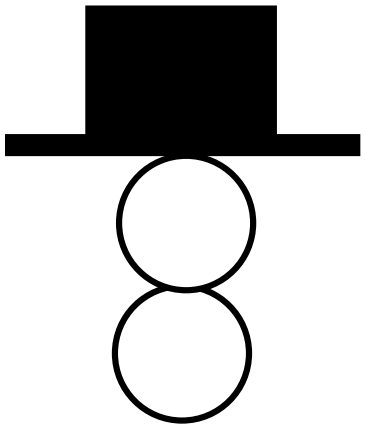
The Snowman Melting



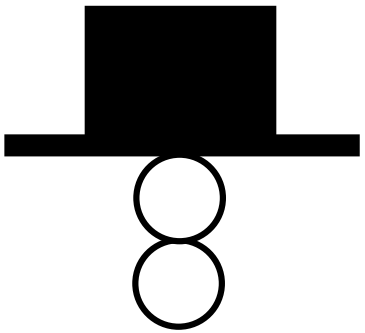
The Snowman Melting



The Snowman Melting



The Snowman Melting

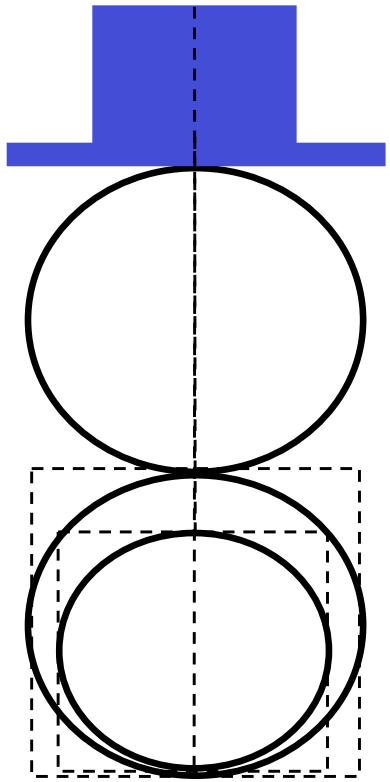


The Composite Pattern

- Define an interface which each component and the entire composite object will implement.
- Sending a melt message to the snowman, will then result in sending a melt message to the components.
- Each component will change as necessary to make the entire snowman melt.

Analysis of melting

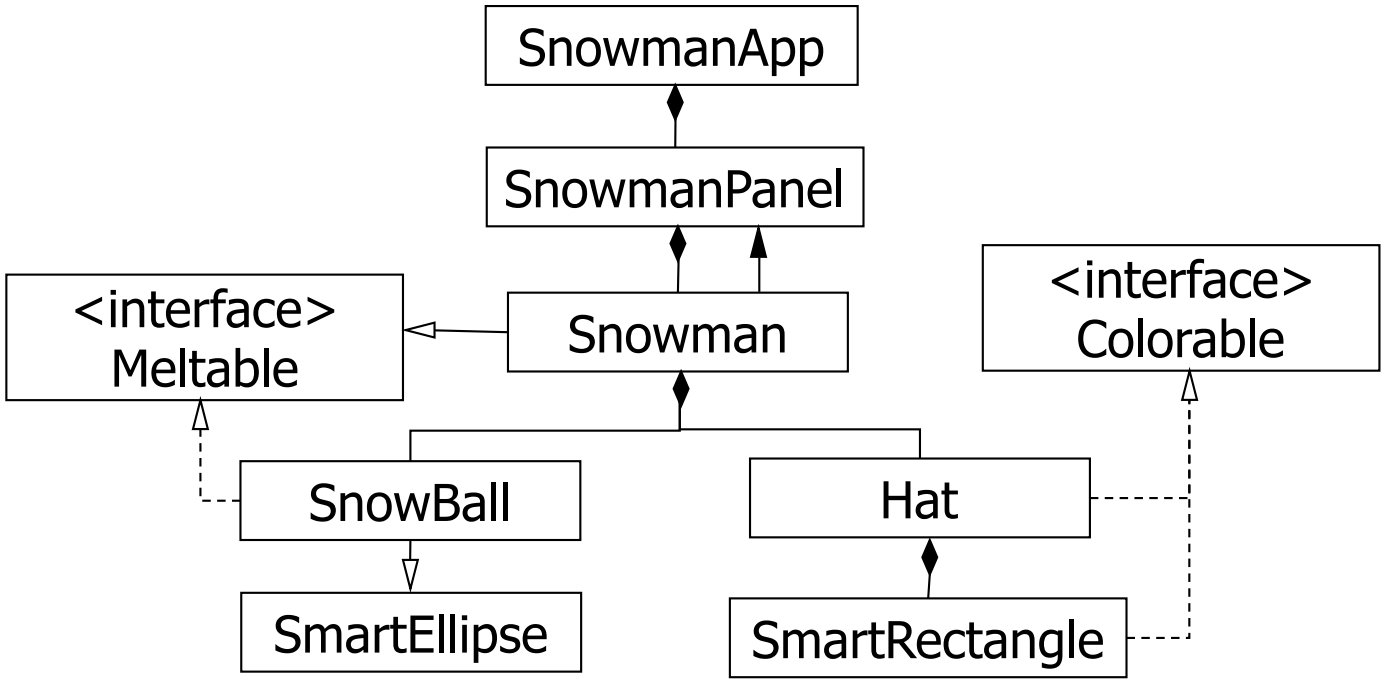
- The bottom snowball decreases its previous width and height by 2*dx pixels moving the x position right dx pixels, and y 2*dx pixels down.
- The upper snowball decreases its previous width and height by 2*dx pixels moving the x position right dx pixels, and y 4*dx pixels down.
- The hat stays the same size moving the y position down 4*dt pixels while not changing the x position. (It is not meltable.)



Interface Meltable

```
interface meltable {
  void melt(int dt);
  void move(int aChangeInX, int aChangeInY);
}
```

Basic Design of Melting Snowman



Where to Place the Timer?

- We could put it in the application
  - One timer for the entire application, which might include multiple meltable objects.
  - It would need to tell the drawing panel to animate itself.
- We could put it in the panel
  - Again one timer for entire panel, which might include multiple meltable objects.
  - It would be able to directly send messages to pertinent objects and knows enough to redraw itself when done with animation changes.

Placing the Timer in the Snowman

- Each snowman in the scene would have its own timer, allowing it to melt at different rates. (Perhaps some are in a more shady spot.)
- The snowman objects need to know which panel they are in to repaint.

SnowmanApp

```
import javax.swing.JFrame;
import java.lang.String;
public class SnowmanApp extends JFrame {
  public SnowmanApp(String title) {
    super(title);
    this.setSize(600,450);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.add(new SnowmanPanel());
    this.setVisible(true);
  }
  public static void main (String[] args) {
    SnowmanApp app = new SnowmanApp("A melting snowman");
  }
}
```

SnowmanPanel

```
import javax.swing.JPanel;
import java.awt.*;
public class SnowmanPanel extends JPanel {
  private Snowman _meltingSnowman;
  public SnowmanPanel() {
    super();
    this.setBackground(Color.BLUE);
    _meltingSnowman = new Snowman(this, 100, 200, 50);
  }
  public void paintComponent(Graphics aBrush) {
    super.paintComponent(aBrush);
    Graphics2D betterBrush = (Graphics2D)aBrush;
    _meltingSnowman.draw(betterBrush);
    _meltingSnowman.fill(betterBrush);
  }
}
```

Snowman

```
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.event.ActionEvent;
import javax.swing.Timer;
import javax.swing.JPanel;
public class Snowman implements Meltable, java.awt.event.Ac
    private SnowBall _upperSnowBall, _lowerSnowBall;
    private Hat _myHat;
    private Timer _myTimer;
    private JPanel _p;
    private final int INTERVAL = 1000;
    private final int MELT_AMOUNT = 2;
    private int _ground;
```

Snowman

```
public Snowman(JPanel panel,int snowBallSize, int X, int Y){
  ground = Y+2*snowBallSize+snowBallSize/2;
  _p = panel;
  _lowerSnowBall = new SnowBall();
  _lowerSnowBall.setSize(snowBallSize, snowBallSize );
  _lowerSnowBall.setLocation(X, _ground - snowBallSize);
  _upperSnowBall = new SnowBall();
  _upperSnowBall.setSize(snowBallSize, snowBallSize);
  _upperSnowBall.setLocation(X, _ground-2*snowBallSize);
  _myHat = new Hat();
  _myHat.setColor(Color.RED);
  _myHat.setSize(snowBallSize, snowBallSize/2);
  _myHat.setLocation(X, Y);
  _myTimer = new Timer(INTERVAL, this);
  _myTimer.start();
  }
```

Snowman

```
public void melt(int dt) {
  _lowerSnowBall.melt(dt);
  _upperSnowBall.melt(dt);
  _upperSnowBall.move(0,(int)(2*dt));
  _myHat.move(0, (int)(4*dt));
  }
  public void move(int aChangeInX, int aChangeInY) {
  int x, y, snowBallRadius, centerHatWidth;
  x = _myHat.getX() + aChangeInX;
  y = _myHat.getY() + aChangeInY;
  _myHat.setLocation(x, y);
  snowBallRadius = ((int)_upperSnowBall.getWidth()/2);
  centerHatWidth = x + (int)(_myHat.getWidth()/2);
  _upperSnowBall.setLocation(centerHatWidth - snowBallRadius,
                    y+(int)_myHat.getHeight());
  snowBallRadius = ((int)_lowerSnowBall.getWidth()/2);
  _lowerSnowBall.setLocation(centerHatWidth - snowBallRadius,
         y+(int)_myHat.getHeight()+_upperSnowBall.getHeight());
}
```

Snowman

```
public void actionPerformed(ActionEvent e) {
  if (((int)_lowerSnowBall.getHeight())<=MELT_AMOUNT)
     _myTimer.stop();
  else {
    this.melt(MELT_AMOUNT);
     _p.repaint();
  }
}
public void draw(Graphics2D aBrush) {
  _myHat.draw(aBrush);
  _upperSnowBall.draw(aBrush);
  _lowerSnowBall.draw(aBrush);
}
public void fill(Graphics2D aBrush) {
  _myHat.fill(aBrush);
  _upperSnowBall.fill(aBrush);
  _lowerSnowBall.fill(aBrush);
  }
}
```

Snowball

```
import java.awt.Color;
public class SnowBall extends SmartEllipse implements Meltable {
  public SnowBall() {
    super(Color.WHITE);
    this.setBorderColor(Color.DARK_GRAY);
  }
  public void melt(int dt) {
    this.setSize((int)this.getWidth()-2*dt, (int)this.getHeight()-
    this.setLocation(this.getX()+dt, this.getY()+2*dt);
  }
}
```

Hat

```
import java.awt.Color;
import java.awt.Graphics2d;
public class Hat implements Colorable {
  private SmartRectangle _brim, _top;
  private int _hatHeight;
  public Hat() {
    _brim = new SmartRectangle(Color.BLACK);
    _top = new SmartRectangle(Color.BLACK);
    _hatHeight = (int)_brim.getHeight() +(int)_top.getHeight();
  }
  public void setColor(Color aColor) {
    _brim.setColor(aColor);
    _top.setColor(aColor);
  }
  public void setBorderColor(java.awt.Color aColor) {
    _brim.setBorderColor(aColor);
    _top.setBorderColor(aColor);
  }
```

Hat

```
public void setFillColor(java.awt.Color aColor) {
  _brim.setFillColor(aColor);
  _top.setFillColor(aColor);
}
public void setSize(int aWidth, int aHeight) {
  int brimHeight, topHeight, topWidth;
  brimHeight = (int)((double)aHeight * 0.2);
  topHeight = aHeight - brimHeight;
  topWidth = (int)((double)aWidth * 0.6);
  _brim.setSize(aWidth, brimHeight);
  _top.setSize(topWidth, topHeight);
}
public void setLocation(int x, int y) {
  _brim.setLocation(x, y + (int)_top.getHeight());
  _top.setLocation(x + (int)((double)(_brim.getWidth()- _top.get
}
```

Hat

```
public void move(int aChangeInX, int aChangeInY) {
  this.setLocation(this.getX()+aChangeInX, this.getY()+aChangeIn
}
public int getX() {
  return (int)_brim.getX();
}
public int getY() {
  return (int)_top.getY();
}
public int getWidth() {
  return (int)_brim.getWidth();
}
```

Hat

```
  public int getHeight() {
    return _hatHeight;
  }
  public void draw(Graphics2D aBrush) {
    _top.draw(aBrush);
    _brim.draw(aBrush);
  }
  public void fill(Graphics2D aBrush) {
    _top.fill(aBrush);
     _brim.fill(aBrush);
  }
}
```