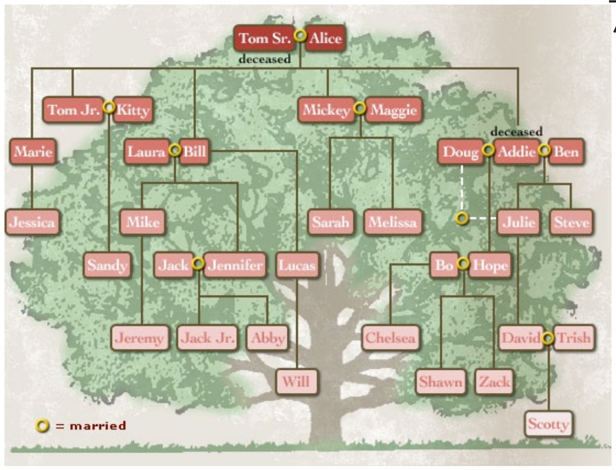
CS415

INTRODUCTION TO COMPUTER SCIENCE

FALL 2017



7 INHERITANCE

CHAPTER 3

CONSTRUCTORS, MUTATORS AND ACCESSORS: REVIEW

- Remember method come in three types:
  - Constructors, used to create new objects:

```
Rectangle rectangle = new Rectangle( )
```

  - Mutators, used to change existing objects:

```
rectangle.setLocation( 50, 50 )
```

  - Accessors, used to access information from existing objects:

```
int x = rectangle.getXLocation( );
```

WRITING CONSTRUCTORS AND MUTATORS

- We have written constructors and Mutators
  - Constructor for Target class:

```
Target t1 = new Target( 10, 50 );
```

```
public Target(int x, int y)
{
   makeTarget( x, y );
}
```

  - Mutator for the Target class

```
t1.move( 5,2 );
```

```
private void move( int dx, int dy )
{
   int x = level1.getXLocation() + dx;
   int y = level1.getYLocation() + dy;
   setLocation( x, y );
}
```

ACCESSORS RETURN VALUES

```
public class Target
{
    private int _size = 60;
```

- Suppose you want to write an accessor for Target that will return its size.
- When you call an accessor you get a value back:

```
int size  = t1.getSize( );
```

- When you write an accessor you must return a value.

```
public int getSize( )
{
    return _size;
}
```

return type

return statement

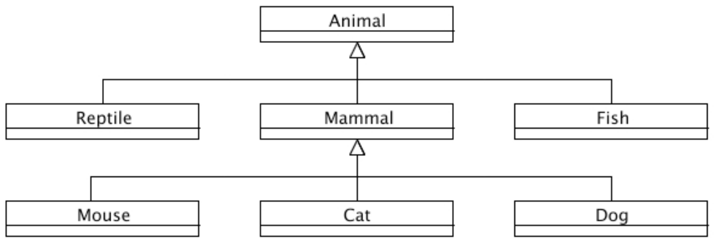
CONSTRUCTORS, MUTATORS AND ACCESSORS

	Using the method	Writing the method
Constructor	"I need to create a new object"	"I must initialize all my instance variables"
Accessor	"I want some information about this object"	"I must return the values of some of my instance variables"
Mutator	"I want this object to change"	" I need to change my instance variables"

ORGANIZING OBJECTS INTO CATEGORIES

- Just as we model objects we will need to model the relationships among the objects
- In every day life we group objects into categories.
  - A pine tree "is a" plant., a car "is a" vehicle.
- Inheritance models the "is-a" relationship.

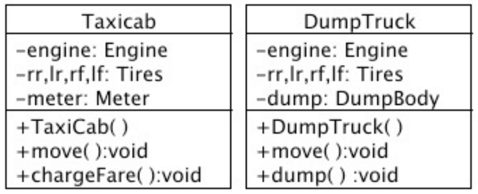
INHERITANCE TREE



- We call this structure a tree, (as in family tree.)
- We say mammal "inherits" from Animal (a mammal "is an" animal).
- Higher in the tree: more general, Lower in the tree: more specialized

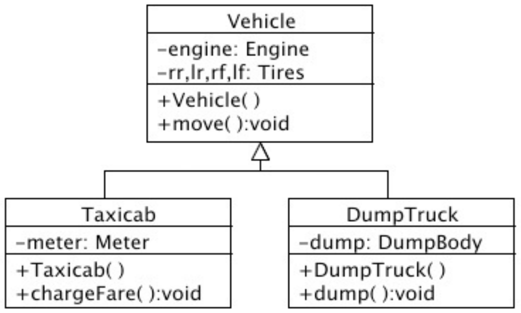
INHERITANCE IN JAVA

- How can this help us in Java?
- Suppose we wanted to write two classes to define Taxicabs and DumpTrucks.



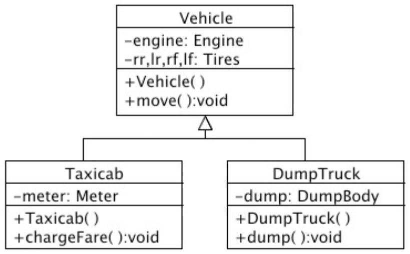
- Much of the code would be duplicated which suggests an inheritance hierarchy.
- The common features are assigned to a new class, Vehicle.

INHERITANCE IN JAVA



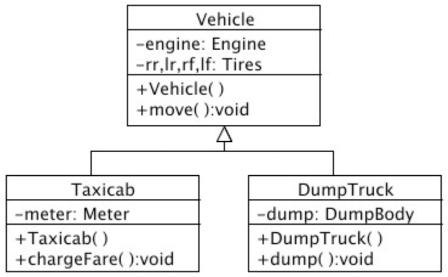
- Vehicle is a superclass of Taxicab and DumpTruck.
  - aka: base class or parent class
- Taxicab and DumpTruck are each a subclass of Vehicle.
  - aka: derived class or child class

INHERITANCE



- A subclass inherits capabilities and properties from its superclass
- A subclass specializes its superclass
  - by *adding* new methods and properties,
  - by *overriding* existing methods,
  - *by defining* “abstract” methods declared by parent
- A superclass *factors out* capabilities common to its subclasses

INHERITANCE



- Subclasses inherit public capabilities (methods).
- Subclasses inherit private properties (instance variables) but do not have access to them.

JAVA INHERITANCE

WITHOUT

```
public class Taxicab
{
   private Engine engine;
   private Wheel rr, lr, rf, lf;
   private Meter meter;

   public Taxicab() { ... }
   public void move() { ... }
   public void chargeFare () { ... }
}

public class DumpTruck
{
   private Engine engine;
   private Wheel rr, lr, rf, lf;
   private DumpBody  dump;

   public DumpTruck() { ... }
   public void move() { ... }
   public void dump () { ... }
}
```

WITH

```
public class Vehicle
{
   private Engine engine;
   private Wheel rr, lr, rf, lf;

   public Vehicle() { ... }
   public void move() { ... }
}

public class Taxicab extends Vehicle
{
   private Meter meter;

   public Taxicab() { ... }
   public void chargeFare () { ... }
}

public class DumpTruck extends Vehicle
{
   private DumpBody  dump;

   public DumpTruck() { ... }
   public void dump () { ... }
}
```

OVERRIDING AN INHERITED METHOD

- We can override an inherited method by redefining it in the derived class.
- Taxicab has overridden its inherited move( ) method with its own version

```
public class Vehicle
{
    private Engine engine;
    private Wheel rr, lr, rf, lf;

    public Vehicle() { ... }
    public void move() { ... }
}

public class Taxicab extends Vehicle
{
    private Meter meter;

    public Taxicab() { ... }
    public void move( ) { ... }
    public void chargeFare () { ... }
}
```

METHOD RESOLUTION

- Methods can be overridden in subclasses. What happens if the move( ) message is sent to a Vehicle?
  - Java searches up the tree for the first definition of a move( ) method.
  - Once it finds one, it executes only that one
    - For a TaxiCab, this will be TaxiCab.move()
    - For a DumpTruck, it will be Vehicle.move()
- This is called method resolution.

OVERLOADING AN INHERITED METHOD

- Just as before we can overload methods.
- Taxicab has (an inherited) move( ) method which is now overloaded by move(int), it now has two methods with that name;

```
public class Vehicle
{
    private Engine engine;
    private Wheel rr, lr, rf, lf;

    public Vehicle() { ... }
    public void move() { ... }
}

 public class Taxicab extends Vehicle
{
    private Meter meter;

    public Taxicab() { ... }
    public void move( int n ) { ... }
    public void chargeFare () { ... }
}
```

OVERLOADING VS OVERRIDING

- Overloading occurs when a class defines two or more methods with the same name but different signatures (parameter lists) --- the class has multiple methods with that name.
- Overriding occurs when a derived class defines a method with the same signature as an inherited method --- the class has only one method with that name and signature.

THE KEYWORD: THIS

- The keyword this refers to the current object.
- Taxicab is overriding the move method, which in turn calls its own startMeter method.

```
public class Taxicab extends Vehicle
{
   private Meter meter;

   public Taxicab() { ... }
   public move( )
   {
      this.startMeter();
      ....
   }
   public void startMeter(){ ... }
   public void chargeFare () { ... }
}
```

```
In most circumstances, this is optional:
   1. If no variable is used to specify the object associated with a
      method invocation, you mean the current object (this).
   2. Same convention applies to references to instance variables.
```

THE KEYWORD SUPER

- The keyword super refers to the superclass of the current class.
- Taxicab overrides the move method, which calls the super class move method.

```
public class Taxicab extends Vehicle
{
   private Meter meter;

   public Taxicab() { ... }
   public void move( )
   {
      this.startMeter();
      super.move();
   }
   public void startMeter(){ ... }
   public void chargeFare () { ... }
}
```

THE DERIVED CLASS CONSTRUCTOR

- As usual, the constructor should initialize the instance variables.
- But how do we initialize our inherited instance variables?
- We just let the super class constructor take care of them

```
public class Taxicab extends Vehicle
{
   private Meter meter;

   public Taxicab()
   {
      super( );
      meter = new Meter( );
   }
   public void move( ){ ... }
   public void startMeter(){ ... }
   public void chargeFare () { ... }
}
```

THE KEYWORD `PROTECTED`

- Although a `Taxicab` has an (inherited) `engine`, it can't access it directly since it is declared as `private` in `Vehicle`. This is called pseudo-inheritance.
- Sometimes a derived class may need to access an inherited variable directly.
- This can be done if the base class declares the variable as protected rather than private

PROTECTED VARIABLES

- Since Vehicle declared engine as protected, Taxicab can directly access it.

```
public class Vehicle
{
   protected Engine engine;
   private Wheel rr, lr, rf, lf;

   public Vehicle() { ... }
   public void move() { ... }
}

public class Taxicab extends Vehicle
{
   private Meter meter;

   public Taxicab() { ... }
   public void move(int n) { ... }
   public void chargeFare ()
   {
      engine.idle();
       ...
   }
}
```

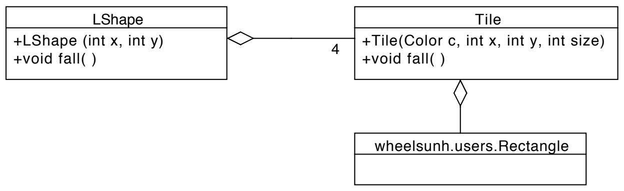
HIERARCHY FOR WHEELSUNH.USERS

- Part of an inheritance tree in wheelsunh

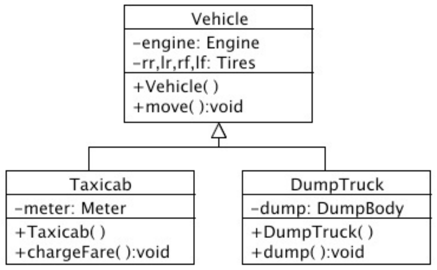
- class wheelsunh.etc.**AbstractGraphic**
  - class wheelsunh.users.**Shape**
    - class wheelsunh.users.**Line**
    - class wheelsunh.users.**RectangularShape**
      - class wheelsunh.users.**Ellipse**
      - class wheelsunh.users.**Rectangle**
      - class wheelsunh.users.**RoundedRectangle**
        - class wheelsunh.users.**ConversationBubble**
  - class wheelsunh.users.**ShapeGroup**

UML RELATIONSHIPS

- "Has-A", Containment:


LShape
+LShape (int x, int y)
+void fall( )
4
Tile
+Tile(Color c, int x, int y, int size)
+void fall( )
wheelsunh.users.Rectangle

- "Is-A", Inheritance:



MOUSE EVENTS FOR WHEELSUNH SHAPES

- A look at the *wheelsunh* API reveals the following methods for *AbstractGraphic*
- Since all the shapes derive from *AbstractGraphic*, they all inherit these methods.

```
void mouseClicked( java.awt.event.MouseEvent e ){ ... }
```
Called when mouse is clicked over the graphic.
```
void mouseDragged( java.awt.event.MouseEvent e ){ ... }
```
Called when mouse button is pressed and dragged over graphic.
```
void mousePressed( java.awt.event.MouseEvent e ) { ... }
```
Called when a mouse button is pressed over the graphic.
```
void mouseReleased( java.awt.event.MouseEvent e ) { ... }
```
Called when a mouse button is released over the graphic.

CLICKABLECIRCLE

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

public class ClickableCircle extends Ellipse
{
    public ClickableCircle(int x, int y)
    {
        super(x,y);
    }

    public void mousePressed( MouseEvent me )
    {
        setColor(java.awt.Color.blue);
    }

    public void mouseReleased( MouseEvent me )
    {
        setColor( java.awt.Color.red );
    }

    public static void main( String [] s)
    {
        new Frame();
        new ClickableCircle( 200, 200 );
    }
}
```

COMPOSITE OBJECTS

- A composite graphical object (such as an *ATV*) shares some basic characteristics with a simple graphical object (like a rectangle):
  - location
  - orientation
- It can also be very convenient for it to share other characteristics:
  - mouse behavior
- wheelsunh supports this in its <u>ShapeGroup</u> class

WHEELSUNH SHAPEGROUP

- *ShapeGroup* models a group of shapes (such as a *ATV* composed of Ellipses, Rectangles, and Lines) that will behave like a single shape.
- *ShapeGroup* has a <u>*setLocation*</u> method that changes the locations of all its components so they are consistent with the group's location
- If we extend *ShapeGroup* we do not need to implement *setLocation* ourselves --- although we may want to.

SHAPEGROUP METHODS

- *ShapeGroup* <u>new</u> method:
  - *add(AbstractShape):*
    - you must add each component shape to the group
- *ShapeGroup* <u>inherited</u> methods:
  - *getXLocation( ), getYLocation( ), setLocation(int x, int y)*
  - *getRotation( ), setRotation(int degrees)*
  - all the *mouse handling* methods
    - Any shape that has been added to the ShapeGroup forwards its mouse events to the group.
    - The group can then implement mouse event handling methods.

TARGET SHAPEGROUP

```
// The “old” way
public class Target
{
   public Target( int x, int y )
   {
      Ellipse outer = new Ellipse(x, y);
      Ellipse inner = new Ellipse(x + 10, y + 10);
            //change sizes and colors of both
   }
   public void setLocation(int x, int y)
   {
      outer.setLocation(x, y);
      inner.setLocation(x + 10, y + 10);
   }
}
```

```
// the new way
public class Target extends ShapeGroup
{
   public Target( int x, int y )
   {
      Ellipse outer  = new Ellipse( 0, 0 );
      this.add( outer );
      Ellipse inner = new Ellipse( 10, 10 );
      this.add( inner );
      // set color and size of both ...


      // ShapeGroup setLocation!
      this.setLocation( x, y );
   }
}
```

CLICKABLEGROUP

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

public class ClickableGroup extends ShapeGroup
{
    private Rectangle rect;
    private Ellipse ellip;

    public ClickableGroup( int x, int y )
    {
        rect = new Rectangle( 0, 0 );
        add(rect);
        ellip = new Ellipse( 50, 0 );
        add(ellip);

        setLocation( x, y );
    }

    public void mousePressed( MouseEvent me )
    {
        rect.setColor(java.awt.Color.blue);
    }

    public void mouseReleased( MouseEvent me )
    {
        rect.setColor( java.awt.Color.red );
    }

    public static void main( String [] s)
    {
        new Frame();
        new ClickableGroup( 200, 200 );
    }
}
```

ABSTRACT CLASSES

- Sometimes a base class is not used to create objects, but simply as a tool to collect the common features of the derived classes.
- For example, we may want TaxiCabs and DumpTrucks in a program, but we will never have generic “vehicles”.
- We can make *Vehicle* abstract.

ABSTRACT CLASSES

- Suppose we want a *dailyReport*( ) method for our vehicles. A *DumpTruck* will report the number of loads and a *Taxicab* will report the meter reading, but a generic *Vehicle* has nothing to report.
- We will put an abstract *dailyReport*() method into *Vehicle*.

ABSTRACT CLASSES

- Abstract Class: The keyword abstract appears after the public modifier in the class definition.
- Abstract method: The keyword abstract appears after the public modifier in the method and the body of the method is replaced with **;**

```
public abstract class Vehicle
{
   protected Engine engine;
   private Wheel rr, lr, rf, lf;

   public Vehicle() { ... }
   public void move() { ... }
   public abstract void dailyReport();
}

public class Taxicab extends Vehicle
{
   private Meter meter;

   public Taxicab() { ... }
   public void move(int n) { ... }
   public void chargeFare () { ... }
}
```

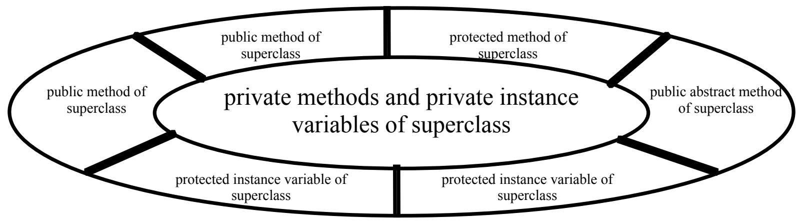
ABSTRACT CLASSES

- Abstract Class: No instances of class Vehicle can be created.
- Abstract method: Every derived class must override the method.

```
public abstract class Vehicle
{
   protected Engine engine;
   private Wheel rr, lr, rf, lf;

   public Vehicle() { ... }
   public void move() { ... }
   public abstract void dailyReport();
}

public class Taxicab extends Vehicle
{
   private Meter meter;

   public Taxicab() { ... }
   public void dailyReport()
   {
       ....
   }
   public void move(int n) { ... }
   public void chargeFare () { ... }
}
```

WHAT CAN BE INHERITED?

- What can a subclass inherit from its superclass?


public method of superclass
protected method of superclass
public method of superclass
private methods and private instance variables of superclass
public abstract method of superclass
protected instance variable of superclass
protected instance variable of superclass

- Only the features in the outer ring!

REVIEW

- Inheritance Hierarchies in Java
- Super class / sub class
- Extending a class
- Overriding inherited methods
- Abstract classes
- Mouse Methods and ShapeGroup

NEXT TIME

- Inheritance helps us to manage complexity by organizing similar classes into an (is-a) hierarchy.
- Sometimes classes can be very different but share some common behavior (a dog can walk and a robot can walk) how will we organize this "acts as" relationship.
- Both classes will implement a *CanWalk* interface.
- Read Chapter 4.