Topic 26
Introduction to Inheritance and Polymorphism

"One purpose of CRC cards [a design tool] is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards that it would be to reorganize a large amount of source code. "

- Cay Horstmann

Based on slides for Building Java Programs by Reges/Stepp, found at http://faculty.washington.edu/stepp/book/

Managing Complexity

- **software development**: The practice of conceptualizing, designing, constructing, documenting, and testing large-scale computer programs. (a.k.a. software engineering)
- **Challenges:**
  - **managing lots of programmers**
  - **dividing work**
  - **avoiding redundant code (wasted effort)**
  - **finding and fixing bugs**
  - **testing**
  - **maintenance (between 50% and 90% of cost)**
- **Code reuse**: writing code once and reusing it in different applications and programs.

Categories

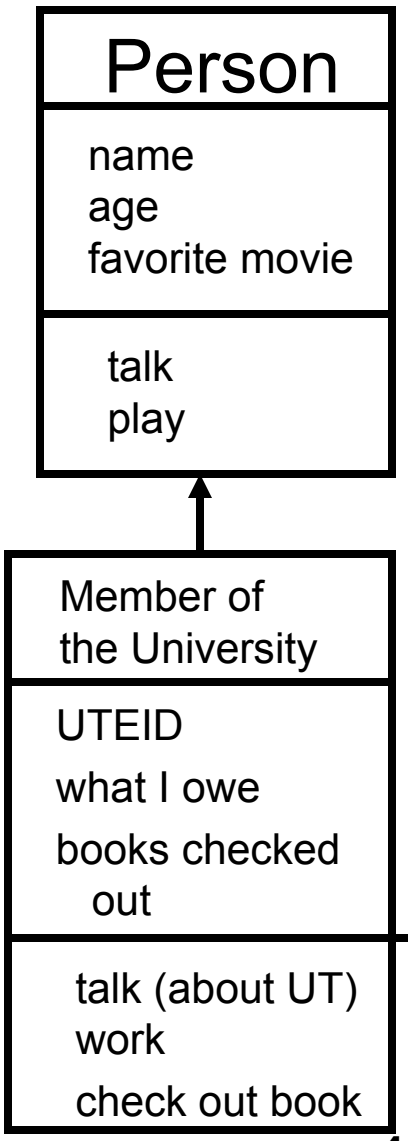
- Often we categorize objects in the real world starting with a general category and then becoming more specific.
  - Let's think about people, human beings. What things does every human being have? What does every human being know how to do?
  - We could draw a diagram of these things. Something like this:

data

behaviors

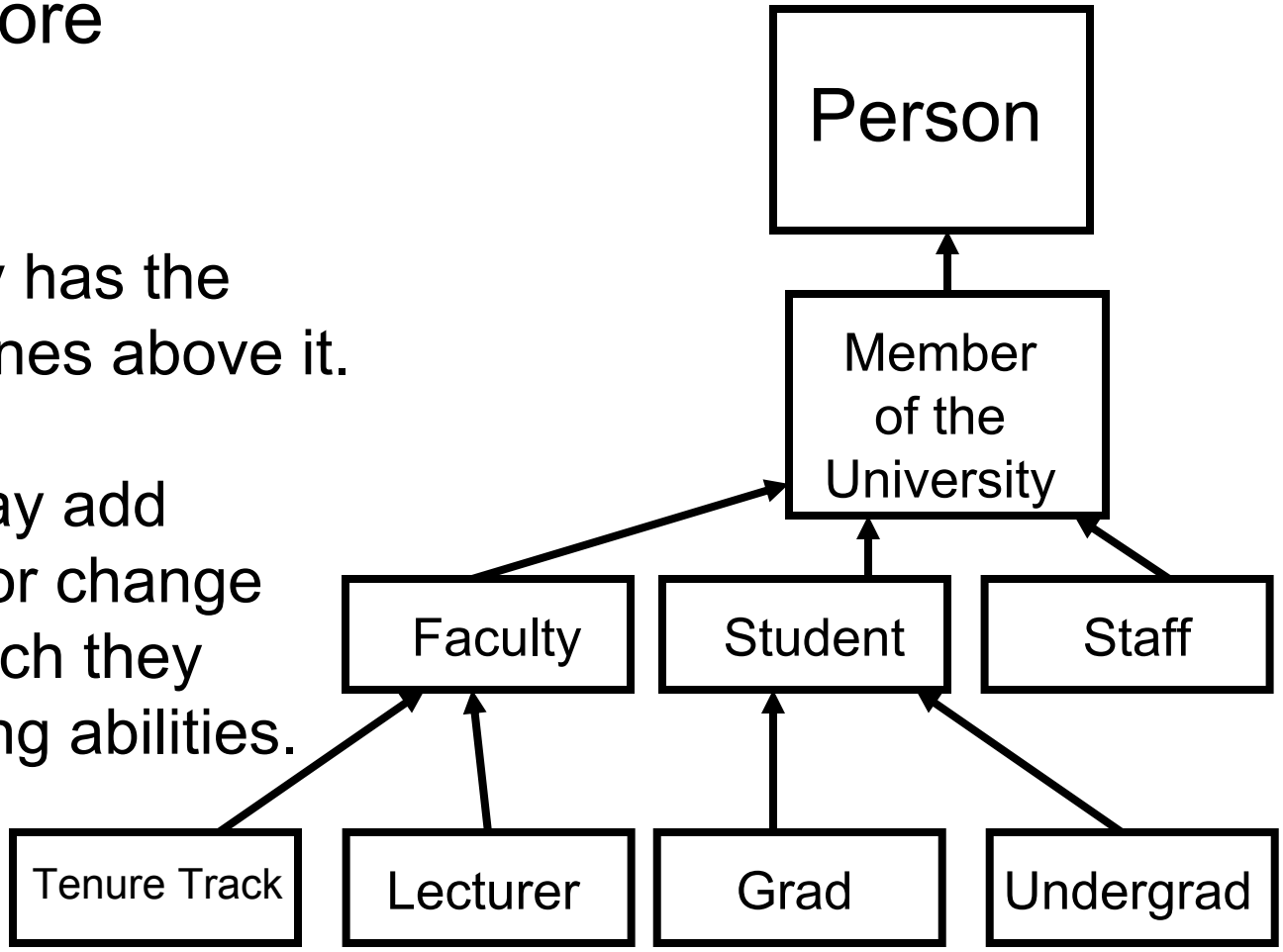
Subcategories

- Within the broad range of people, let's talk about a specific group:
  members of the University
  - What new attributes and behaviors are there?
  - Let's assume that all members are people, and draw them as a subcategory of persons, as shown to the right
  - Members of the University add some new abilities and attributes.
    - UTEID
    - work
  - Employees perform some of the original person's abilities differently.
    - talk about the University
  - Notice we don't **repeat** things from Person



More categorization

- We can add more categories.
  - Each category has the ability of the ones above it.
  - Categories may add new abilities, or change the way in which they perform existing abilities.



Categories as code

- Let's implement a class for a person

```
public class Person {

    private String favoriteMovie;

    public void talk(){
         System.out.println("Hi!");
    }

    public void play(){
         System.out.println("Watching " +
              favoriteMovie);
    }

}
```

Subcategory, first try

- The class for Member has much in common with Person:

```
public class UniversityMember {

    private String favoriteMovie;
    private String UTEID;

    public void talk(){
        System.out.println("Hi!");
    }

    public void play(){
```

Inheritance

- **inherit**: To gain all the data fields and methods from another class, and become a child of that class.
  - **superclass**: The class from which you inherit.
  - **subclass**: The class that inherits.

- Inheritance, general syntax:

```
public class <name> extends <class name> {
```

  - Example:

```
public class UniversityMember
                              extends Person {
```

  - Each member object now automatically:
    - has a talk and play method
    - can be treated as a Person by any other code
      (e.g. an Employee could be stored as an element of a Person[])

Subclass with inheritance

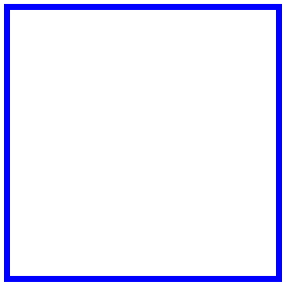
- A better version of the UniversityMember class:

```
public class UniversityMember extends Person{

    private String UTEID;


      public void who(){
              System.out.println("My UTEID is " + UTEID);
      }
}
```

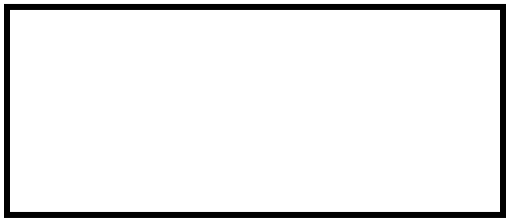
  - We only write the portions that are unique to each (data) type.
  - If we have a UniversityMember object we can call all methods from Person and UniversityMember

Another example: Shapes

- Imagine that we want to add a class Square.



square

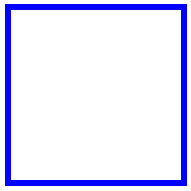


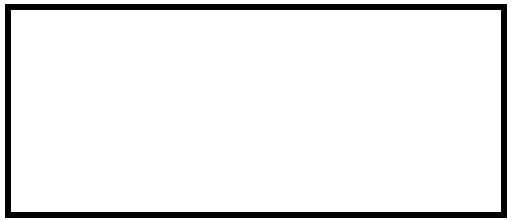
rectangle

  - A square is just a rectangle with both sides the same length.
  - If we wrote Square from scratch, its behavior would largely duplicate that from Rectangle.
  - We'd like to be able to create a Square class that absorbs behavior from the Rectangle class, to remove the redundancy.

Square and Rectangle

- Differences between square code and rectangle code:
  - Square only needs one size parameter to its constructor.
    `Square sq = new Square(3, 7, 4);` rather than
    `Rectangle rect = new Rectangle(2, 9, 13, 6);`
  - Squares print themselves differently with toString.
    `"Square[x=3,y=7,size=4]"` rather than
    `"Rectangle[x=2,y=9,width=13,height=6]"`





Inheritance

- Reminder: inheritance, general syntax:

```
public class <name> extends <class name> {
```

- Let's declare that all squares are just special rectangles:

```
public class Square extends Rectangle {
```

- Each Square object now automatically:
  - has an x, y, width, and height data field
  - has a getArea, getPerimeter, draw, toString, intersection method
  - can be treated as a Rectangle by any other code (e.g. a Square could be stored as an element of a Rectangle[])

Inheritance and constructors

- We want Squares to be constructed with only an (x, y) position and a size (because their width and height are equal).
  - But internally, the Square has x/y/width/height data fields, which all need to be initialized.
  - We can make a Square constructor that uses the Rectangle constructor to initialize all the data fields.
- Syntax for calling superclass's constructor:

  `super(` ***<parameter(s)>*** `);`

  - Example:

```
public class Square extends Rectangle {
    public Square(int x, int y, int size) {
        super(x, y, size, size);
    }
}
```

    - Each Square object is initialized with its width and height equal. The size parameter to the Square constructor provides the value for the width and height parameters to the superclass Rectangle constructor.

Overriding methods

- We don't want to inherit the toString behavior, because Squares print themselves differently than rectangles.
- **override**: To write a new version of a method in a subclass, replacing the superclass's version.
  - If Square declares its own toString method, it will replace the Rectangle toString code when Squares are printed.

```
public String toString() {
    return "Square[x=" + this.getX() + ",y=" + this.getY() +
        ",size=" + this.getWidth() + "]";
}
```

    - We have to say, for example, this.getX() instead of this.x because the data fields are private (can only actually be modified inside the Rectangle class).
    - We'll use this.getWidth() as our size, but the height would also work equally well.

Relatedness of types

- We've previously written several 2D geometric types such as Circle and Rectangle.
  - We could add other geometric types such as Triangle.

- There are certain attributes or operations that are common to all shapes.
  - perimeter - distance around the outside of the shape
  - area - amount of 2D space occupied by the shape

- Every shape has these attributes, but each computes them differently.

Shape area, perimeter

- Rectangle
  - area = $w\ h$
  - perimeter = $2w + 2h$
- Circle
  - area = $\pi\ r^2$
  - perimeter = $2\ \pi\ r$
- Triangle
  - area = $(1/2)\ b\ h$
  - perimeter = $side1 + side2 + side3$

Common behavior

- Let's write methods getPerimeter, getArea, and draw for all our shape types.

- We'd like to be able to treat different shapes in the same way, insofar as they share common behavior, such as:
  - Write a method that prints any shape's area and perimeter.
  - Create an array of shapes that could hold a mixture of the various shape objects.
  - Return a method that could return a rectangle, a circle, a triangle, or any other shape we've written.
  - Make a DrawingPanel display many shapes on screen.

Interfaces

- **interface**: A type that consists only of a set of methods, that classes can pledge to implement.
  - Interfaces allow us to specify that types have common behavior, by declaring that they implement a common interface.
  - The interface type can be used to refer to an object of any type that implements the interface's methods.
  - A method may accept a parameter of an interface type, or return a value of an interface type. In these cases, a value of any implementing type may be passed / returned.

Interface syntax

- Let's declare that certain types of objects can be Shapes, and that every Shape type has a getArea and getPerimeter method.

```
public interface Shape {
    public double getArea();
    public double getPerimeter();
    public void draw(Graphics g);
}
```

- Interface declaration, general syntax:

```
public interface <name> {
    <method header(s)> ;
}
```

  - A method header specifies the name, parameters, and return type of the method.
  - The header is followed by a `;` and not by `{ }`
    - The implementation is not specified, because we want to allow each shape to implement its behavior in its own way.

Implementing an interface

- **implement** an interface: To declare that your type of objects will contain all of the methods in a given interface, so that it can be treated the same way as any other type in the interface.

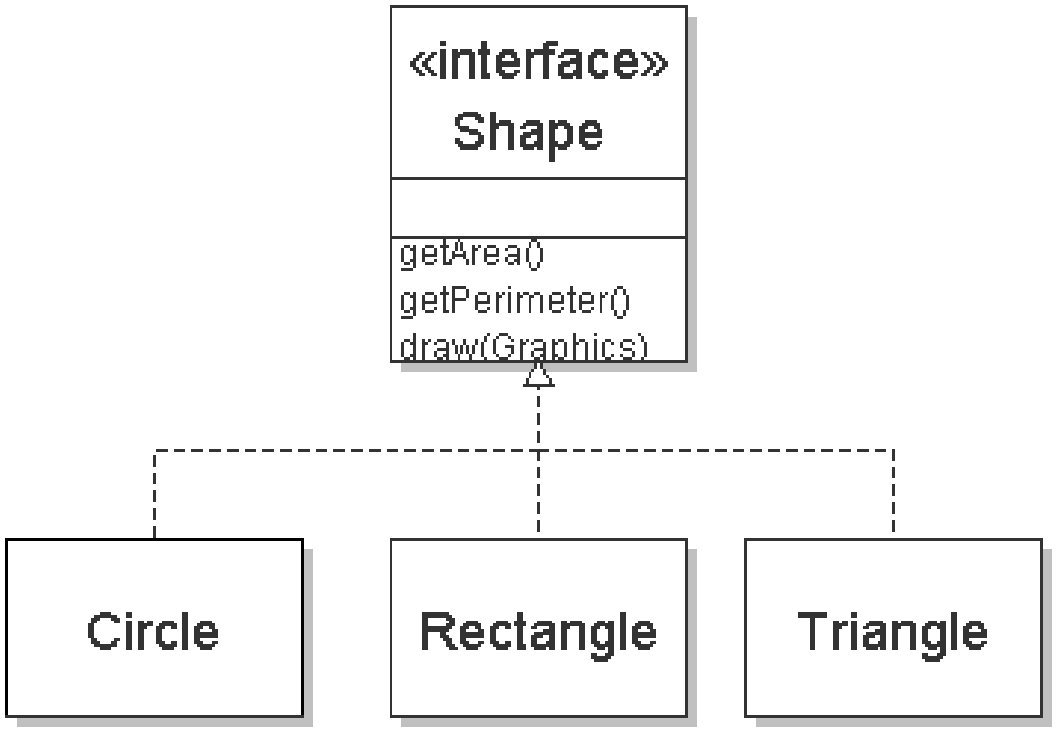
- Implementing an interface, general syntax:

```
public class <class name> implements <interface name>
{
    ...
}
```

  - Example: We can specify that a class is a Shape.

```
public class Circle implements Shape {
    ...
}
```

Diagrams of interfaces



- We draw arrows upward from the classes to the interface(s) they implement.
  - There is an implied superset-subset relationship here; e.g., all Circles are Shapes, but not all Shapes are Circles.
  - This kind of picture is also called a *UML class diagram*.

Interface requirements

- Since the Shape interface declares a getArea and getPerimeter method, this mandates that any class wishing to call itself a Shape must have these two methods.

- If we write a class that claims to be a Shape but doesn't have both of these methods, it will not compile.
  - Example:

```
public class Banana implements Shape {

}

C:\foo\Banana.java:1: Banana is not abstract and
does not override abstract method getArea() in
Shape
public class Banana implements Shape {
             ^
```

How interfaces are used

- We can write a method that accepts an object of an interface type as a parameter, or returns an interface type.
  - Example:

```
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.print("area: " + s.getArea());
    System.out.println(", perimeter: " +
                       s.getPerimeter());
    System.out.println();
}
```

  - Any object that implements the interface may be passed as the parameter to this method.

- **polymorphism**: The ability to run the same code on objects of many different types.

Using polymorphism

- The following code uses the printInfo method from the previous slide, with shapes of 2 different types.

```
public static void main(String[] args) {
    Circle circ = new Circle(new Point(3, 7), 6);
    Rectangle rect = new Rectangle(10, 20, 4, 7);
    printInfo(circ);
    printInfo(rect);
}
```

Output:

```
The shape: Circle[center=(3, 7),radius=6]
area: 113.09733552923255, perimeter:
  37.69911184307752

The shape: Rectangle[x=10,y=20,width=4, height=7]
area: 28.0, perimeter: 22.0
```

Arrays of interface type

- We can create an array of an interface type, and store any object implementing that interface as an element.

```
Circle circ = new Circle(new Point(3, 7), 6);
Rectangle rect = new Rectangle(10, 20, 4, 7);
Triangle tri = new Triangle(new Point(),
                  new Point(7, 9), new Point(15, 12));
Shape[] shapes = {circ, tri, rect};
for (int i = 0; i < shapes.length; i++) {
    printInfo(shapes[i]);
}
```

  - Each shape executes its appropriate behavior when it is passed to the printInfo method, or when getArea or getPerimeter is called on it.