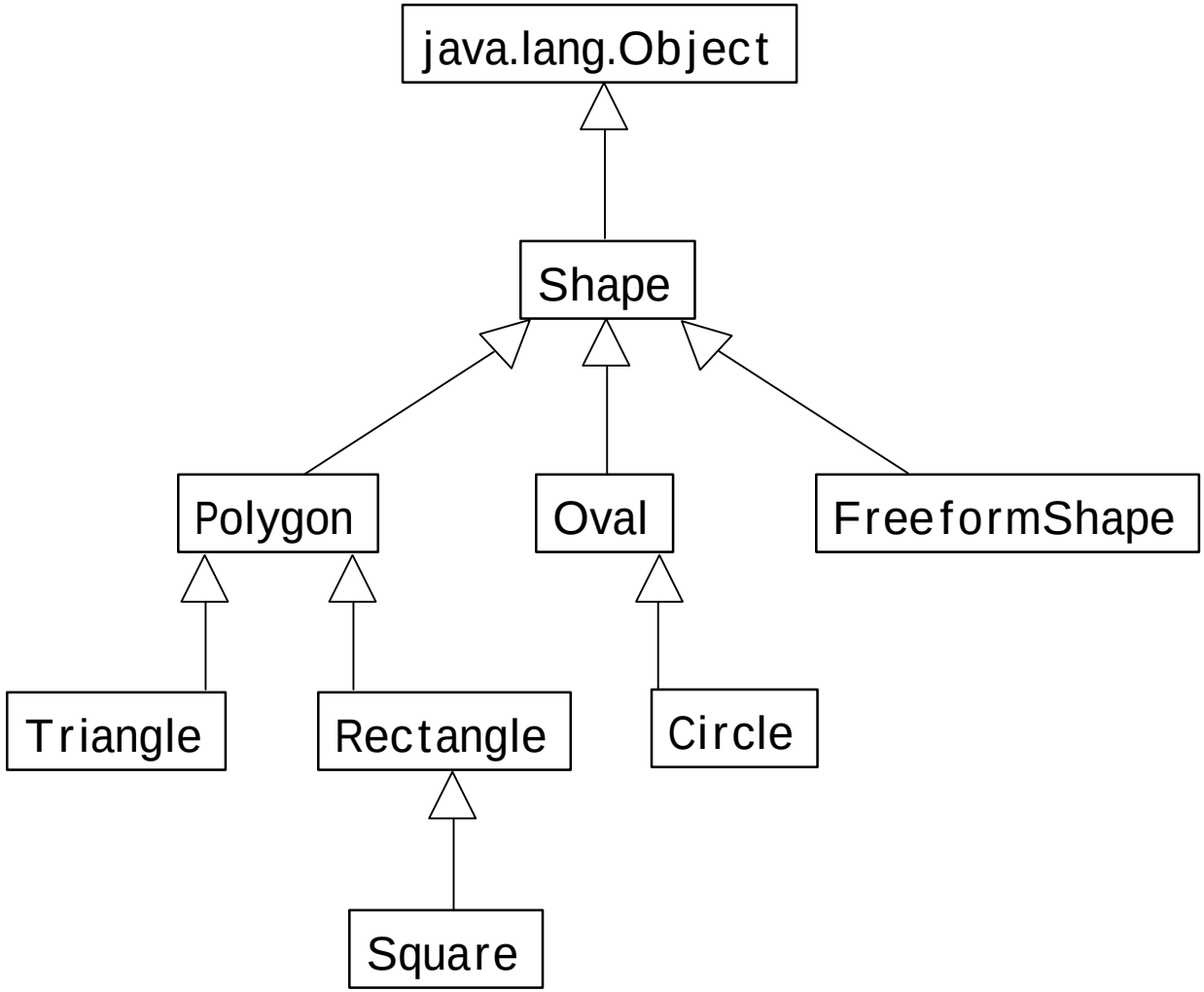
Object-Oriented Programming and Design

Part II

Object-based vs. object-oriented programming

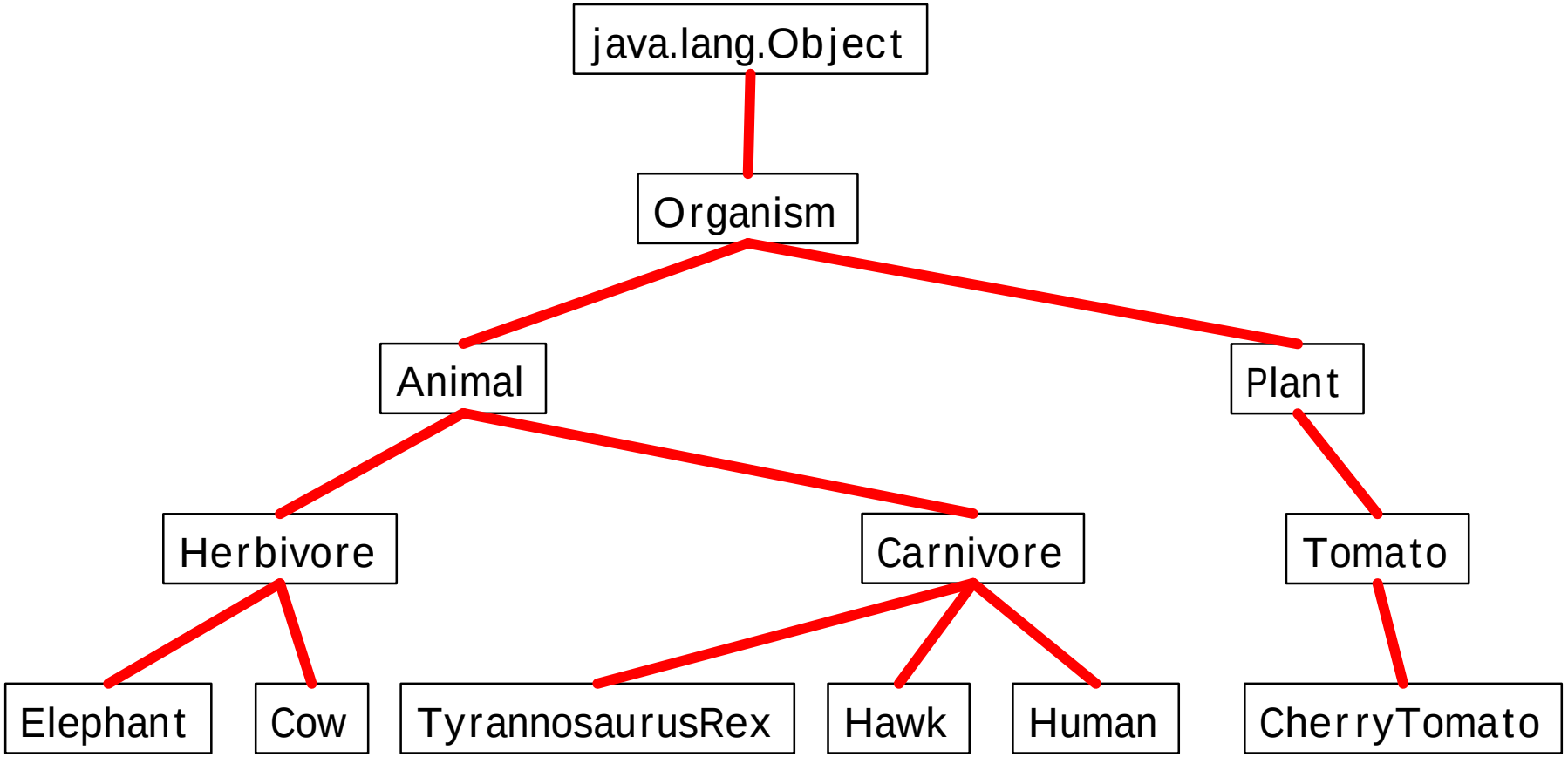
- So far, we did mostly object-based programming
  - Classes, objects, instantiating objects, this
  - calling methods
  - has-a and part-of relationships (i.e., fields)
- Object-oriented programming = object-based +
  - Subclass, extends, superclass, super(), protected
  - Assignment of subclass object to superclass var
    i.e. casting
  - Inheritance or is-a hierarchies
  - Abstract classes and methods
  - Polymorphism (i.e., calling generic methods)
  - Method overriding in a subclass (i.e., method in a subclass with the same name)
  - Inheritance hierarchies are used to express commonality, abstraction and facilitate reuse

Inheritance, Is-a, Class Hierarchy



Data Modeling: Result Inheritance Quiz

- Arrange the classes below into an inheritance hierarchy



Inheritance Relationships review

- Subclass
  - *extends* a *superclass* definition with new fields or methods
  - *inherits* the fields and methods of the superclass
  - modifies the meaning of the *superclass*
  - forms an *is-a* relationship with its superclass
- A subclass inherits both data (fields) and behavior (methods)
  - inherited members can be accessed as if they were present in the subclass itself
  - Subclasses have access to the public, protected and package members of its superclasses
  - Subclass methods **and** other methods of other classes in the same package have access to protected members
  - constructors and private members are not inherited

Inheriting and Extending -- review

- Overriding a superclass method
  - A subclass can redefine a superclass method by using the *same signature*
- Overloading of a method
  - A method in the same class or a subclass with the same name but *different signature*

Overriding and Overloading

Subclass member kind	Name	Argument types and return type	Effect
instance method	same	same	overrides
instance method	same	different	overloads
static method	same	same	hides
static method	same	different	extends
instance or static method	different	any	extends
instance or static field	same		hides
instance or static field	different		extends

super, this

- This and super are references
- The keyword **super** refers to the parent class within which **super** appears
- The keyword **this** refers to the object of the class within which **this** appears

Two kinds of method overriding

- Replacement
  - A method completely replaces the method of the superclass that is overridden (e.g., a toString() routine in every subclass).
- Refinement
  - The superclass method is not replaced but rather refined, that is, code is added to the superclass method. This is accomplished by first calling the superclass method (e.g., super.abc())
  - All subclass constructors use the refinement method. This is called constructor chaining. Each subclass constructor begins its execution by first calling its superclass constructor (i.e., super())
    - There is one exception to this….

Interactive programming exercise: MyClass2.java

Inheritance Quiz

- For each member of class B, state the effect of each member, that is, overrides, overloads, hides, or extends

```
class A {
   protected String name;
   public static int getCount() {return 1;}
   public String toString() {return name;}
   private void doStuff() { … }
   public Object getStuff() { … }
}

class B extends A {
   public StringBuffer name = new StringBuffer(toString());
   public static int getCount() {return 2;}
   public String toString(String suffix) {
      name.append(suffix);  return name.toString();
   }
   public void doStuff() { … }
   protected String getStuff() { … }
}
```

Type Polymorphism

- a method declared in a superclass is overridden in the subclass
- you have an instance of the subclass that did the override, but it's referenced by a variable of the type of the superclass…
- Which actual method implementation will be called?

```
class Shape {
   public String toString() {
      return "Shape";
   }
}
```

```
class Oval extends Shape {
   public String toString() {
      return "Oval";
   }
}
```

```
public static void main(String[] args) {
    Shape shape = new Shape();
    Oval oval = new Oval();
    shape.toString(); // prints Shape
    oval.toString(); // prints Oval
    shape = oval;
    shape.toString(); // prints ?
    }
```

Interactive programming exercises: Oval.java; Wind2.java

Polymorphism – why?

- Polymorphism deals with decoupling in terms of types.
- We already saw how inheritance allows the treatment of an object as its own type or its base type.
- This allows many types (derived from the same base type) to be treated as if they were one type, and a single piece of code to work on all those different types equally.
- The polymorphic method call allows one type to express its distinction from another, similar type, as long as they're both derived from the same base type.
- Polymorphism is possible because of late or dynamic binding
- Polymorphism allows us to more easily extend our programs – it allows us to separate the things that will change in our programs to the things that won't change

Subtyping and Substitutability

- Whenever an instance of a class is expected, you can *always* substitute an instance of one of its descendants

```
Shape s = new Rectangle(); // ok
Circle c = new Circle(); // ok
s = c; //ok
s = doMagic(c); //ok
```

```
Shape doMagic(Shape s) {
    …
    return new Square();
}
```

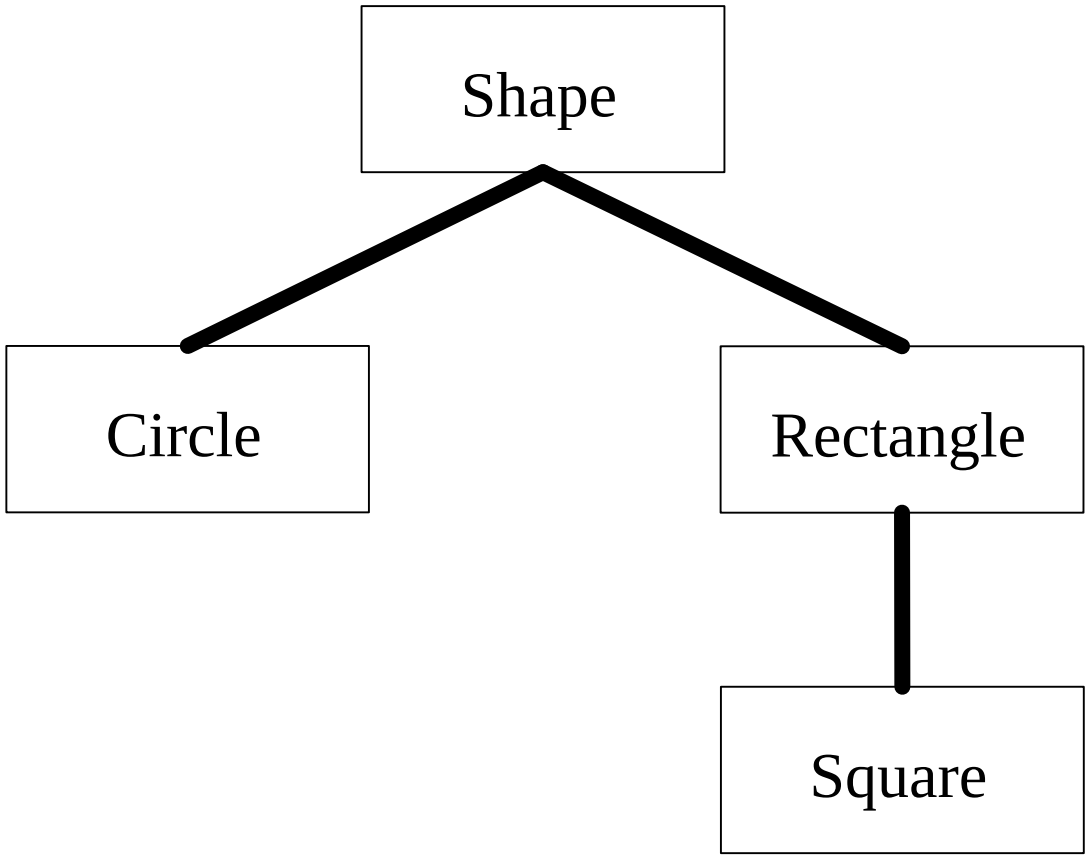
- But you *cannot* substitute an instance of one of its ancestors, or of an unrelated class

```
Rectangle r = new Polygon(); //run-time error
r = new Circle(); //run-time error
r = doMagic(r); //run-time error
```

Overriding Details -- summary

- Constructors
  - are not inherited
  - in a subclass, every constructor must call a superclass constructor as its first operation
    - called **constructor chaining**
    - **super( );** is usually (implicitly) called first in every subclass constructor
- Regular methods
  - Overridden methods can completely replace the super class's method or can refine the method by calling **super.method(**arguments**)** within the subclass method
- **static** methods
  - Cannot be overridden, but can be hidden
- **abstract** methods
  - *Must* be overridden unless the subclass is also abstract

Classic shape inheritance hierarchy



Class Shape -- revisited

```
public abstract class Shape {

    // forces all subclasses to implement a method area()
    public abstract double area();
    public abstract double circumference();

    // toString() can be overriden by subclasses;
    // toString() could also be declared abstract;
    // if a subclass does not implement a toString()
    // method, then it will output "Shape"
    public String toString() {  return "Shape";  }

}
```

Class Circle

```
//this class has no toString() method
public class Circle extends Shape {
    protected int r;

    public Circle(int r) { super(); this.r = r; }

    public double area() { return r*r*Math.PI; }

    public double circumference() { return (r+r)*Math.PI; }
}
```

Class Rectangle

```
public class Rectangle extends Shape {
    protected int width;
    protected int height;

    // constructor
    public Rectangle(int width, int height) {
        this.width = width; this.height = height;
    }
    public double area() { return width*height; }
    public double circumference() {
        return width+width + height + height;
    }

    // override the toString() method of Shape
    public String toString() { return "Rectangle"; }

}
```

Class Square

```
public class Square extends Rectangle {

    public Square(int side) { super(side, side); }    // calls Rectangle's
                                                      // constructor

    public double area() { return width*width; }

    public String toString() { return "Square"; }

}
```

Class Geometry

```
public class Geometry {

    public static void main(String[] args) {
        Shape[] s = new Shape[10]; // holds any shapes
        // an object of a subclass can be treated as an object of its superclass
        s[0] = new Circle(5); s[1] = new Circle(10); s[2] = new Circle(20);
        s[3] = new Circle(30); s[4] = new Rectangle(10,20);
        s[5] = new Rectangle(5, 10); s[6] = new Rectangle(3, 4);
        s[7] = new Square(10); s[8] = new Square(20); s[9] = new Square(5);
        for (int k=0; k<s.length; k++) {
          // polymorphism, dynamic method binding
          System.out.println(k + " " + s[k].toString() + " a=" +
            (int)(s[k].area()) + " c=" + (int)(s[k].circumference()));
        }
    }
}
```

Output produced by main() in class Geometry

```
0 Shape a=78 c=31
1 Shape a=314 c=62
2 Shape a=1256 c=125
3 Shape a=2827 c=188
4 Rectangle a=200 c=60
5 Rectangle a=50 c=30
6 Rectangle a=12 c=14
7 Square a=100 c=40
8 Square a=400 c=80
9 Square a=25 c=20
```

Casting

- What if you want to substitute an instance of what *looks* like an ancestor, but you *know* is really a descendant?

```
Shape s     = new Rectangle();
Rectangle r =  (Rectangle)s; // we can do this!
Square q    =  (Square)s;  // what about this?
```

- You must explicitly state that the instance is actually of a substitutable type
  - this is called *casting* (or, more specifically, *downcasting*)
  - this can fail at compile-time if what you state is completely impossible:

```
Square q = (Square) new Circle(); // error
```

  - usually, your statement is checked at runtime; if it's wrong, a **ClassCastException** is thrown

Casting Quiz

- Insert appropriate casts where needed, mark invalid statements:

```
Shape   shape;
Square  square;
Oval    oval;

oval  =             new Oval();
shape =             oval;
square  =           oval;
square  =           shape;
shape =             new Square();
square  =           shape;

Holder h =          new Holder();
h.set(               square);
shape =             h.get();
oval  =             h.get();
h.set(              h.get());
```

```
class Holder {
   private Object o;
   Holder() {}
   void set(Object o) {
      this.o = o;
   }
   Object get() {
      return o;
   }
}
```

Casting Quiz

- Insert appropriate casts where needed, mark invalid statements:

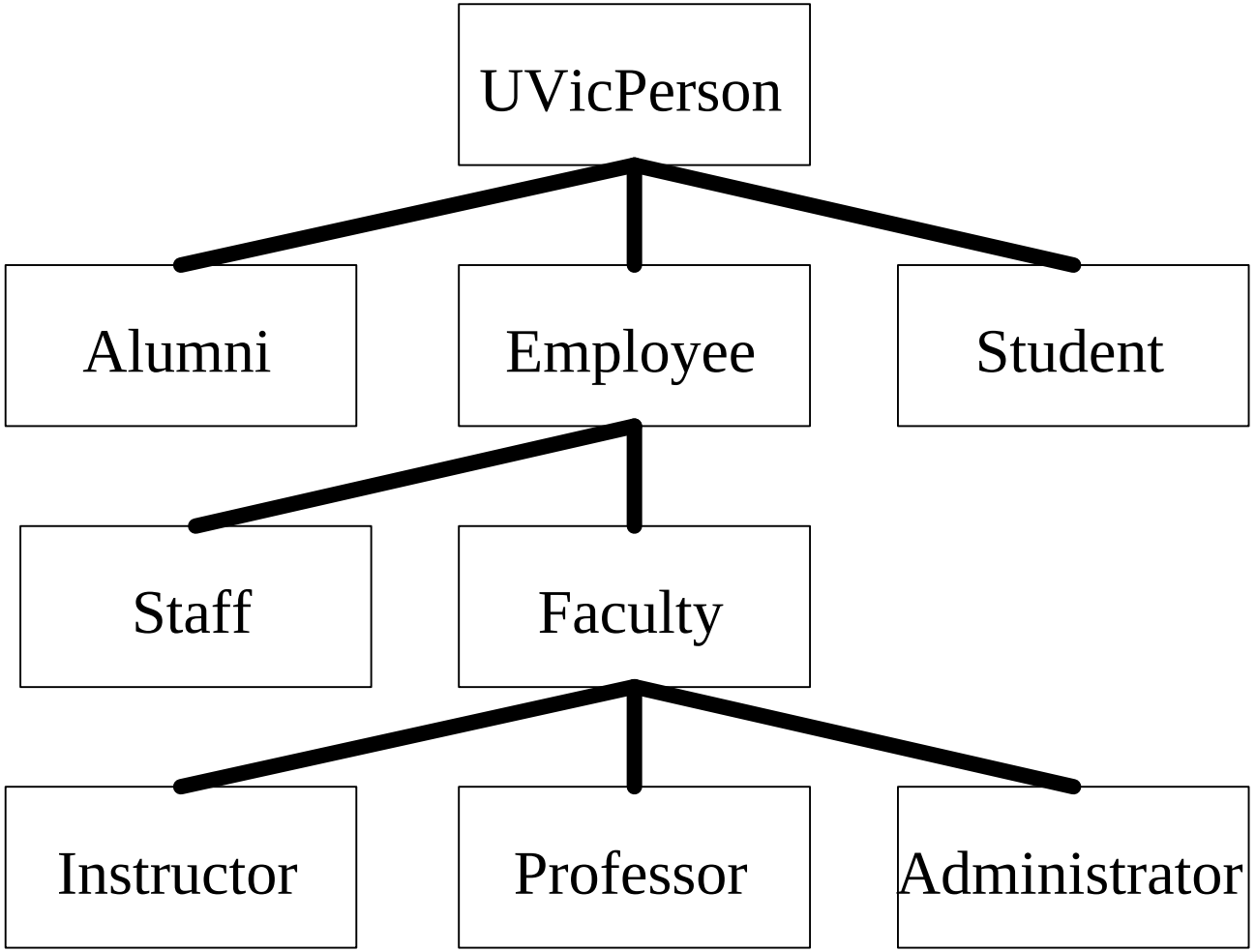
```
Shape   shape;
Square  square;
Oval    oval;

oval  =            new Oval(); // ok
shape =            oval; // ok
square  =          oval; // error
Square  =          shape; // error
shape =            new Square(); // ok
square  = (Square) shape; //cast

Holder h = new Holder();
h.set(             square   );
shape =  (Shape)   h.get(); //cast
oval  =            h.get(); // error
h.set(             h.get());
```

```
class Holder {
   private Object o;
   Holder() {}
   void set(Object o) {
      this.o = o;
   }
   Object get() {
      return o;
   }
}
```

UVic inheritance hierarchy



Class UVicPerson

```
public abstract class UVicPerson {
    protected String first;
    protected String last;
    protected long id;

    public UVicPerson(String last, String first) {
        this.last = last; this.first = first;
        this.id = (long)(Math.random()*1000000);
    }

    public String toString() {
        return " " + last + ", " + first + ", " + id;
    }
}
```

Class Employee

```
public abstract class Employee extends UVicPerson {
    protected double salary;
    public Employee(String last, String first) {
        super(last, first);
    }
    public Employee(String last, String first, double salary) {
        super(last, first);
        this.salary = salary;
    }
    public String toString() {
        return "" +
        super.toString() + ", " +
        (int)salary;
    }
    public abstract void work();
}
```

Classes Faculty and Staff

```
public abstract class Faculty extends Employee {
    public Faculty(String last, String first) {
        super(last, first);
    }
    public Faculty(String last, String first, double salary) {
        super(last, first, salary);
    }
}

public class Staff extends Employee {
    public Staff(String last, String first) {
        super(last, first);
    }
    public Staff(String last, String first, double salary) {
        super(last, first, (double)(Math.random()*100000));
    }
    public void work() { System.out.println("I admin"); }
}
```

Classes Administrator and Professor

```
public class Administrator extends Faculty {
public Administrator(String last, String first) {
        super(last, first);
    }
    public void work() {
        System.out.println("I admin");
    }
}

public class Professor extends Faculty {
    public Professor(String last, String first) {
        super(last, first);
        salary = (double)(Math.random()*100000);
    }
    public Professor(String last, String first, double salary) {
        super(last, first, salary);
    }
    public void work() {
        System.out.println("I research, teach and admin");
    }
}
```

Class Instructor

```
public class Instructor extends Faculty {
    public Instructor(String last, String first) {
        super(last, first);
        salary = (double)(Math.random()*100000);
    }
    public Instructor(String last, String first, double salary) {
        super(last, first, salary);
    }
    public void work() { System.out.println("I teach and admin"); }
}
```

Classes Student and Alumni

```
public class Student extends UVicPerson {
    public Student(String last, String first) {
        super(last, first);
    }
}

public class Alumni extends UVicPerson {
    public Alumni(String last, String first) {
        super(last, first);
    }
}
```

Class UVicCommunity

```
public class UVicCommunity {

    public static void main(String[] args) {
        UVicPerson[] p = new UVicPerson[100];
        p[0] = new Student("Smith", "John"); p[1] = new Student("Carlson", "Brian");
        p[2] = new Student("Gannon", "David"); p[3] = new Student("Cuche", "Didier");
        p[4] = new Alumni("Gosling", "Richard"); p[5] = new Alumni("Parnas", "Gene");
        p[6] = new Professor("Muller", "Hausi"); p[7] = new Professor("Stege", "Ulrike");
        p[8] = new Professor("Ellis", "John"); p[9] = new Instructor("Bultena", "Bette");
        for (int k=0; k<10; k++) {
          // polymorphism, generically process
          // all objects of a class hierarchy
          System.out.println(p[k].toString());
          // determining the actual type of the object
          if (p[k] instanceof Employee) {
            // dynamic method binding
            ((Employee)p[k]).work(); // cast to Employee
          }
        }
    }
}
```

Output produced by main() in class UVicCommunity

```
Smith, John, 511250
Carlson, Brian, 2446
Gannon, David, 205344
Cuche, Didier, 945873
Gosling, Richard, 569145
Parnas, Gene, 662656
Muller, Hausi, 985427, 83142
I research, teach and admin
Stege, Ulrike, 192916, 44089
I research, teach and admin
Ellis, John, 475706, 15197
I research, teach and admin
Bultena, Bette, 437523, 87410
I teach and admin
```

Abstract classes and methods

- An abstract class may contain abstract methods
- An abstract method is a method with no body (i.e., simply a semicolon after the parameter list)
- An abstract method constitutes a protocol or contract, that is, regular or non-abstract subclasses are required to implement the abstract methods of superclasses
- Thus, if a superclass has an abstract method, it guarantees that all subclasses (even future subclasses) implement this method
- For example, an abstract toString() method in a class forces all its subclasses to implement a toString() method

Interactive programming exercises: Wind2.java revisited