

Polymorphism

Object Orientated Programming in Java

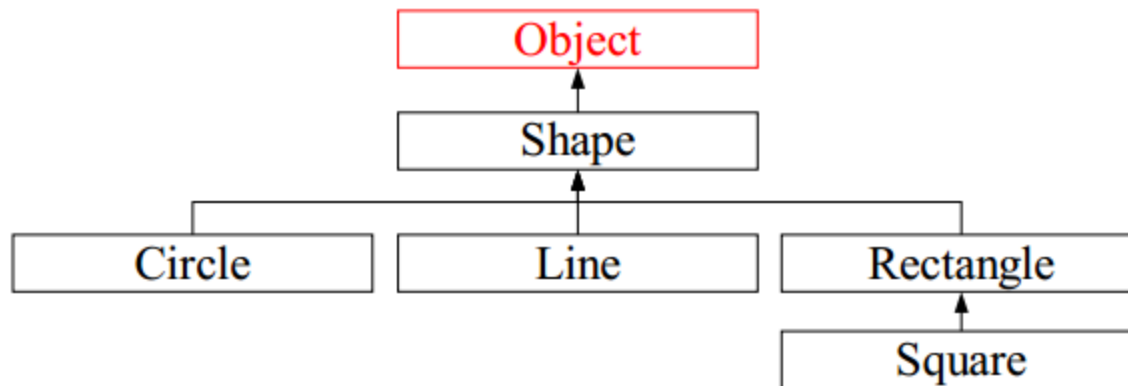
Benjamin Kenwright

Outline

- Why use polymorphism?
- Upcast (and downcast)
- Static and dynamic type
- Dynamic binding
- Polymorphism
 - ▷ A polymorphic field (the state design pattern) Today's Practical
- Review/Discussion
- Summary

Review Class Hierarchies in Java

- Class Object is the root of the inheritance hierarchy in Java
- If no superclass is specified a class inherits implicitly from **Object**
- If a superclass is specified explicitly the subclass will inherit indirectly from Object



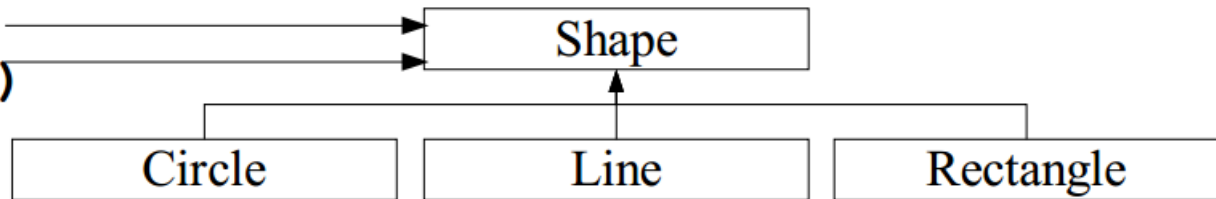
Why Polymorphism?

// substitutability

Shape s;

s.draw()

s.resize()

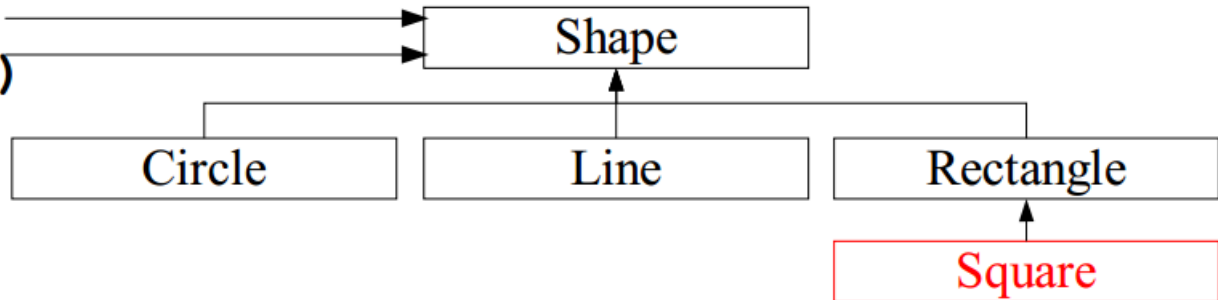


// extensibility

Shape s;

s.draw()

s.resize()



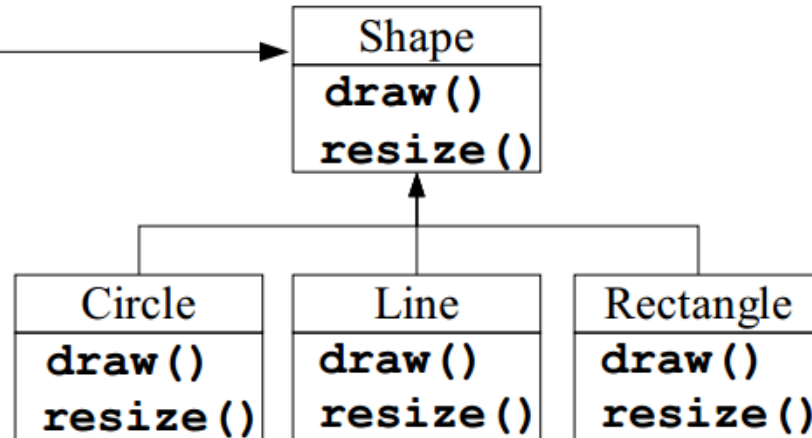
Why Polymorphism?, cont.

```
// common interface
```

```
Shape s;
```

```
s.draw()
```

```
s.resize()
```



Upcast

Downcast

```
// upcasting
```

```
Shape s = new Line();
```

```
s.draw()
```

```
s.resize()
```

Advantages of Upcasting

■ Advantages

- ▷ Code is simpler to write (and read)
- ▷ Uniform interface for clients, i.e., type specific details only in class code, not in the client code
- ▷ Change in types in the class does not effect the clients
- ▷ If type change within the inheritance hierarchy
- ▷ Popular in object-oriented programs
 - Many upcast to Object in the standard library

Disadvantages of Upcasting

■ Disadvantages

- ▷ Must explicitly downcast if type details needed in client after object has been handled by the standard library (very annoying sometimes).

```
Shape s = new Line();
```

```
Line l = (Line) s; // downcast
```

Static and Dynamic Type

- The static type of a variable/argument is the declaration type
- The dynamic type of a variable/argument is the type of the object the variable/argument refers to

```
class A{
    // body
}
class B extends A{
    // body
}
public static void main(String args[]){
    A x;           // static type A
    B y;           // static type B

    x = new A();   // dynamic type A
    y = new B();   // dynamic type B
    x = y;         // dynamic type B
}
```


Polymorphism, informal

- In a bar you say “I want a beer!”
 - ▷ What ever beer you get is okay because your request was very generic
- In a bar you say “I want a Samuel Adams Cherry Flavored beer!”
 - ▷ If you do not exactly get this type of beer you are allowed to complain
- In chemistry they talk about polymorph materials as an example
 - ▷ H₂O is polymorph (ice, water, and steam).

Polymorphism

- *Polymorphism*: “The ability of a variable or argument to refer at run-time to instances of various classes”

```
Shape s = new Shape();  
Circle c = new Circle();  
Line l = new Line();  
Rectangle r = new Rectangle();
```

```
s = l;           // is this legal?  
l = s;           // is this legal?  
l = (Line)s      // is this legal?
```

- The assignment `s = l` is legal if the static type of `l` is `Shape` or a subclass of `Shape`.
- This is *static type checking* where the type comparison rules can be done at compile-time.
- Polymorphism is constrained by the inheritance hierarchy

Dynamic Binding

```
class A {  
    void doSomething() {  
        ...  
    }  
}
```

```
class B extends A {  
    void doSomething () {  
        ...  
    }  
}
```

```
A x = new A();
```

```
B y = new B();
```

```
x = y;
```

```
x.doSomething(); // on class A or class B?
```

- Binding: Connecting a method call to a method body
- Dynamic binding: The dynamic type of x determines which method is called (also called late binding)
 - ▷ Dynamic binding is not possible without polymorphism.
- Static binding: The static type of x determines which method is called (also called early binding)

Dynamic Binding, Example

```
class Shape {
    void draw() { System.out.println ("Shape"); }
}
class Circle extends Shape {
    void draw() { System.out.println ("Circle"); }
}
class Line extends Shape {
    void draw() { System.out.println ("Line"); }
}
class Rectangle extends Shape {
    void draw() {System.out.println ("Rectangle"); }
}
public static void main(String args[]){
    Shape[] s = new Shape[3];
    s[0] = new Circle();
    s[1] = new Line();
    s[2] = new Rectangle();
    for (int i = 0; i < s.length; i++){
        s[i].draw(); // prints Circle, Line, Rectangle
    }
}
```

Polymorphism and Constructors

```
class A { // example from inheritance lecture
    public A(){
        System.out.println("A()");
        // when called from B the B.doStuff() is called
        doStuff();
    }
    public void doStuff(){System.out.println("A.doStuff()"); }
}
class B extends A{
    int i = 7;
    public B(){System.out.println("B()");}
    public void doStuff(){System.out.println("B.doStuff() " + i);
    }
}

//prints
A()
B.doStuff() 0
B()
B.doStuff() 7

public class Base{
    public static void main(String[] args){
        B b = new B();
        b.doStuff();
    }
}
```

Polymorphism and private Methods

```
class Shape {
    void draw() { System.out.println ("Shape"); }
    private void doStuff() {
        System.out.println("Shape.doStuff()");
    }
}

class Rectangle extends Shape {
    void draw() {System.out.println ("Rectangle"); }
    public void doStuff() {
        System.out.println("Rectangle.doStuff()");
    }
}

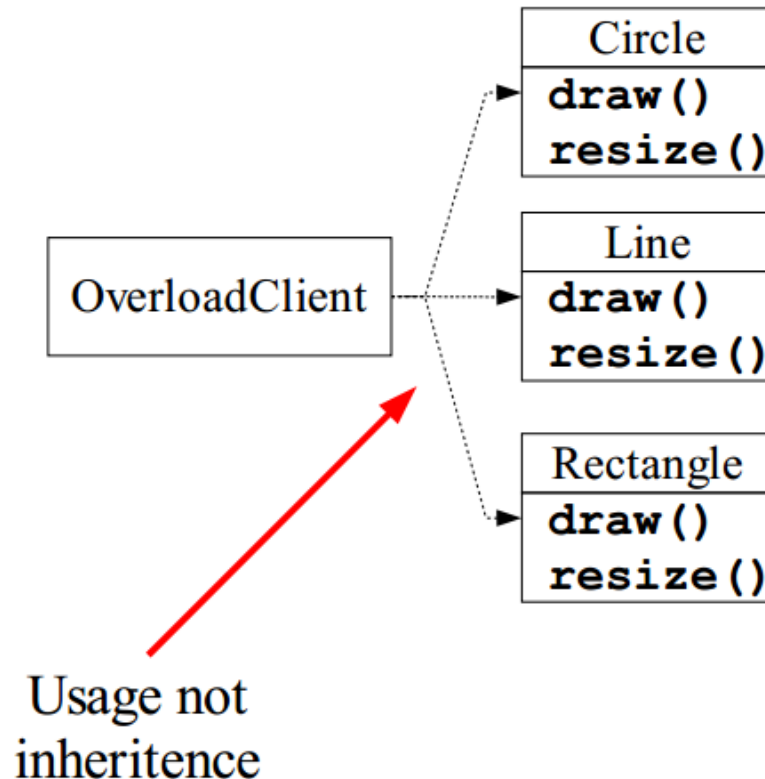
public class PolymorphShape {
    public static void polymorphismPrivate(){
        Rectangle r = new Rectangle();
        r.doStuff();    // okay part of Rectangle interface
        Shape s = r;    // up cast
        s.doStuff();    // not allowed, compiler error
    }
}
```

Why Polymorphism and Dynamic Binding?

- Separate interface from implementation
 - ▷ What we are trying to achieve in object-oriented programming!
- Allows programmers to isolate type specific details from the main part of the code
 - ▷ Client programs only use the method provided by the Shape class in the shape hierarchy example.
- Code is simpler to write and to read
- Can change types (and add new types) with this propagates to existing code

Overloading vs. Polymorphism (1)

- Has not yet discovered that the Circle, Line and Rectangle classes are related.
(not very realistic but just to show the idea)



Overloading vs. Polymorphism (2)

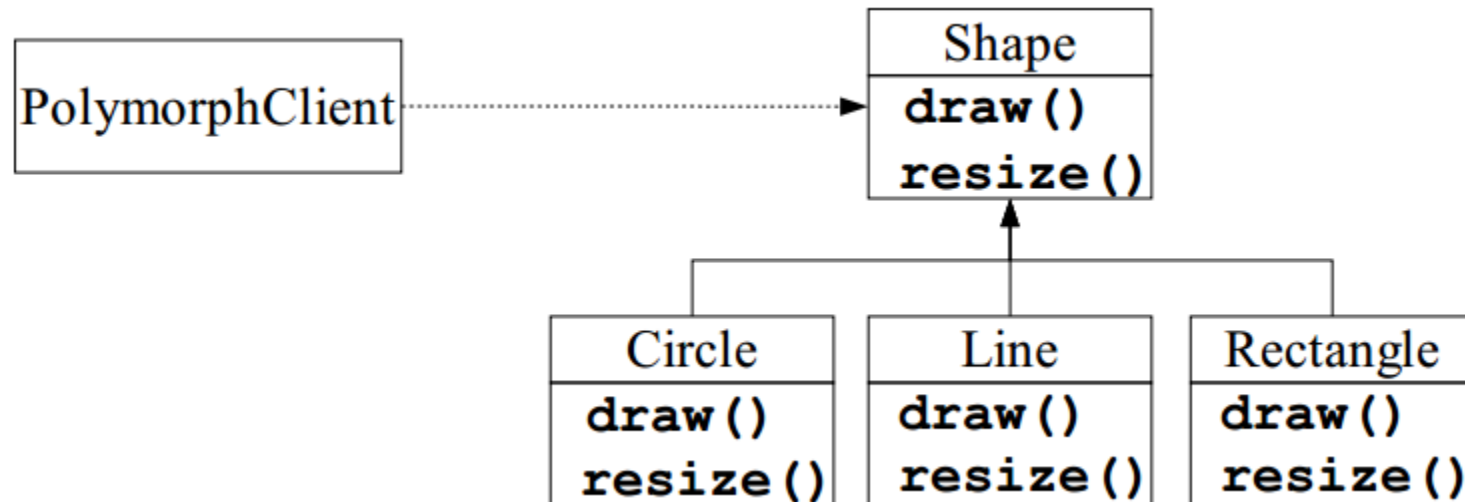
```
class Circle {
    void draw() { System.out.println("Circle"); }}
class Line {
    void draw() { System.out.println("Line"); }}
class Rectangle {
    void draw() { System.out.println("Rectangle"); }}

public class OverloadClient{
    // make a flexible interface by overload and hard work
    public void doStuff(Circle c){ c.draw(); }
    public void doStuff(Line l){ l.draw(); }
    public void doStuff(Rectangle r){ r.draw(); }

    public static void main(String[] args){
        OverloadClient oc = new OverloadClient();
        Circle ci = new Circle();
        Line li = new Line();
        Rectangle re = new Rectangle();
        // nice encapsulation from client
        oc.doStuff(ci); oc.doStuff(li); oc.doStuff(re);
    }
}
```

Overloading vs. Polymorphism (3)

- Discovered that the Circle, Line and Rectangle class are related via the general concept Shape
- Client only needs access to base class methods



Overloading vs. Polymorphism (4)

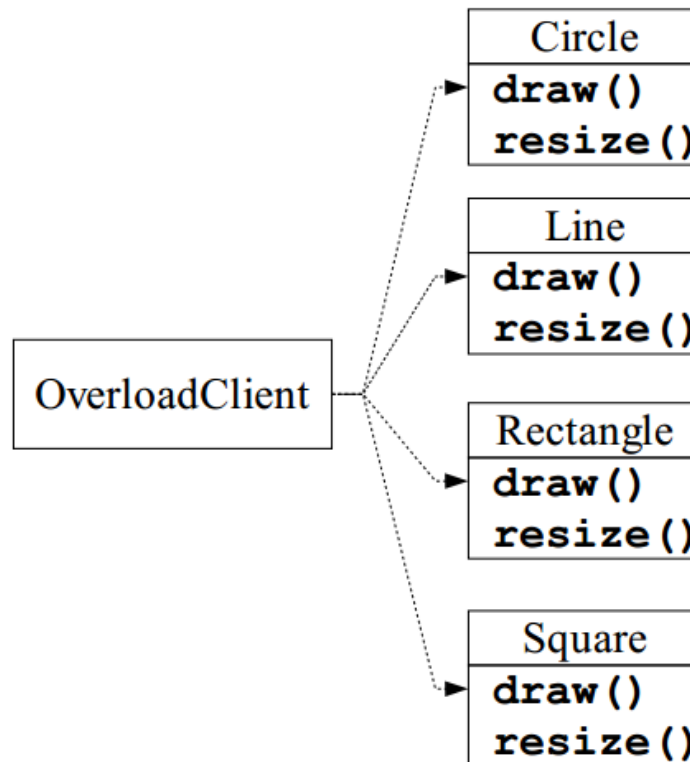
```
class Shape {
    void draw() { System.out.println("Shape"); }
class Circle extends Shape {
    void draw() { System.out.println("Circle"); }}
class Line extends Shape {
    void draw() { System.out.println("Line"); }}
class Rectangle extends Shape {
    void draw() { System.out.println("Rectangle"); }}

public class PolymorphClient{
    // make a really flexible interface by using polymorphism
    public void doStuff(Shape s){ s.draw(); }

    public static void main(String[] args){
        PolymorphClient pc = new PolymorphClient();
        Circle ci = new Circle();
        Line li = new Line();
        Rectangle re = new Rectangle();
        // still nice encapsulation from client
        pc.doStuff(ci); pc.doStuff(li); pc.doStuff(re);
    }
}
```

Overloading vs. Polymorphism (5)

- Must extend with a new class Square and the client has still not discovered that the Circle, Line, Rectangle, and Square classes are related



Overloading vs. Polymorphism (6)

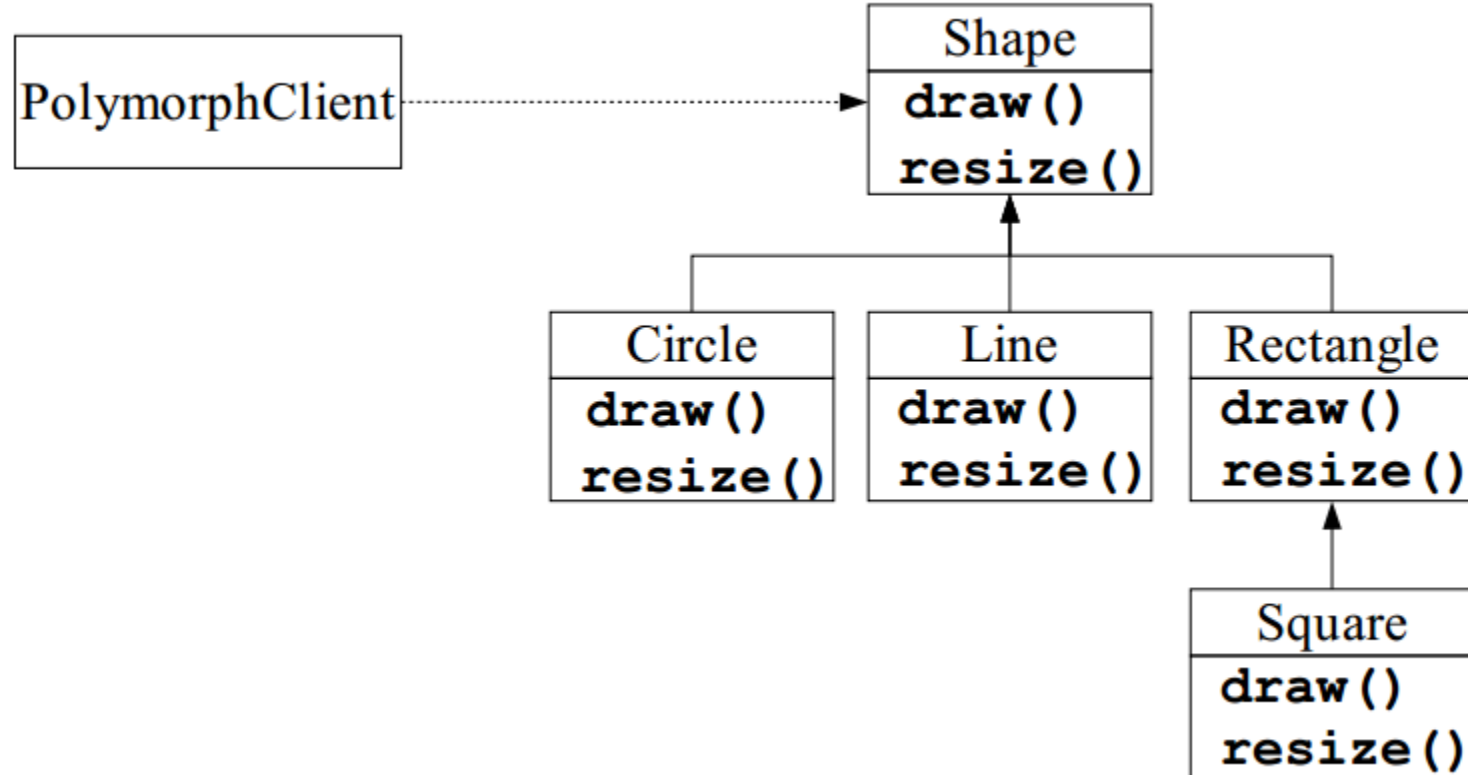
```
class Circle {
    void draw() { System.out.println("Circle"); }
class Line {
    void draw() { System.out.println("Line"); }
class Rectangle {
    void draw() { System.out.println("Rectangle"); }
class Square {
    void draw() { System.out.println("Square"); }

public class OverloadClient{
    // make a flexible interface by overload and hard work
    public void doStuff(Circle c){ c.draw(); }
    public void doStuff(Line l){ l.draw(); }
    public void doStuff(Rectangle r){ r.draw(); }
    public void doStuff(Square s){ s.draw(); }

    public static void main(String[] args){
        <snip>
        // nice encapsulation from client
        oc.doStuff(ci); oc.doStuff(li); oc.doStuff(re);
    }
}
```

Overloading vs. Polymorphism (7)

- Must extend with a new class Square that is a subclass to Rectangle



Overloading vs. Polymorphism (8)

```
class Shape {
    void draw() { System.out.println("Shape"); }
class Circle extends Shape {
    void draw() { System.out.println("Circle"); }}
class Line extends Shape {
    void draw() { System.out.println("Line"); }}
class Rectangle extends Shape {
    void draw() { System.out.println("Rectangle"); }}
class Square extends Rectangle {
    void draw() { System.out.println("Square"); }}

public class PolymorphClient{
    // make a really flexible interface by using polymorphism
    public void doStuff(Shape s){ s.draw(); }

    public static void main (String[] args){
        <snip>
        // still nice encapsulation from client
        pc.doStuff(ci); pc.doStuff(li); pc.doStuff(re);
    }
}
```

The Opened/Closed Principle

■ Open

- ▷ The class hierarchy can be extended with new specialized classes

■ Closed

- ▷ The new classes added do not affect old clients
- ▷ The superclass interface of the new classes can be used by old clients

■ This is made possible via

- ▷ Polymorphism
- ▷ Dynamic binding

Abstract Class and Method

- An abstract class is a class with an abstract method.
- An abstract method is method with out a body, i.e., only declared but not defined.
- It is not possible to make instances of abstract classes.
- Abstract method are defined in subclasses of the abstract class

Abstract Classes in Java

```
abstract class ClassName {  
    // <class body>  
}
```

- ❑ Classes with abstract methods must declared abstract
- ❑ Classes without abstract methods can be declared abstract
- ❑ A subclass to a concrete superclass can be abstract
- ❑ Constructors can be defined on abstract classes.
- ❑ Instances of abstract classes cannot be made
- ❑ Abstract fields not possible

Abstract Class in Java, Example

```
public abstract class Stack{

    abstract public void push(Object el);
    abstract public void pop(); // note no return value
    abstract public Object top();
    abstract public boolean full();
    abstract public boolean empty();
    abstract public int size();
    public void toggleTop(){
        if (size() >= 2){
            Object topEl1 = top(); pop();
            Object topEl2 = top(); pop();
            push(topEl1); push(topEl2);
        }
    }
    public String toString(){
        return "Stack";
    }
}
```

Abstract Methods in Java

```
abstract [access modifier] return type  
                                methodName([parameters]);
```

- A method body does not have to be defined
- Abstract methods are overwritten in subclasses
- Idea taken directly from C++
 - ▷ pure virtual function
- You are saying: “The object should have this properties I just do not know how to implement the property at this level of abstraction.”

Abstract Methods in Java, Example

```
public abstract class Number {  
    public abstract int intValue();  
    public abstract long longValue();  
    public abstract double doubleValue();  
    public abstract float floatValue();  
    public byte byteValue(){  
        // method body  
    }  
    public short shortValue(){  
        // method body  
    }  
}
```

This Week

- Read Associated Chapters
- Review Slides
- Java Exercises
- Online Quizzes

Summary

- Polymorphism an object-oriented “switch” statement.
- Polymorphism should strongly be preferred over overloading
 - ▷ Must simpler for the class programmer
 - ▷ Identical (almost) to the client programmer
- Polymorphism is a prerequisite for dynamic binding and central to the object-oriented programming paradigm.
 - ▷ Sometimes polymorphism and dynamic binding are described as the same concept (this is inaccurate).
- Abstract classes
 - ▷ Complete abstract class no methods are abstract but instantiation does not make sense.
 - ▷ Incomplete abstract class, some method are abstract

Questions/Discussion