Object-Oriented Programming CSE-703029

Faculty of Computer Science

Phenikaa University

Lecture 5: Polymorphism

Slides adapted from Prof. Steven Roehrig

Polymorphism

Means “many forms”

Many classes related to each other by inheritance

Today’s Topics

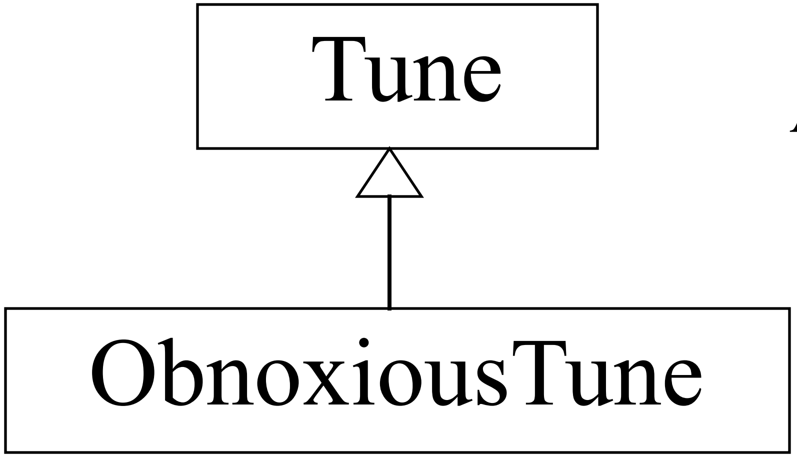
- Upcasting again
- Method-call binding
- Why polymorphism is good
- Constructors and polymorphism
- Downcasting
- Several digressions: the **Object** class, object wrappers, the **Class** class, reflection

Upcasting Again

```
class CellPhone {
    cellPhone() { //…}
    public void ring(Tune t) { t.play(); }
}
class Tune {
    public void play() {
        System.out.println("Tune.play()");
    }
}
class ObnoxiousTune extends Tune{
    ObnoxiousTune() { // …}
    // …
}
```

An **ObnoxiousTune** “is-a” **Tune**

```
public class DisruptLecture {
    public static void main(String[] args) {
        CellPhone noiseMaker = new CellPhone();
        ObnoxiousTune ot = new ObnoxiousTune();
        noiseMaker.ring(ot);  // ot works though Tune called for
    }
}
```



A “UML” diagram

Aspects of Upcasting

- Upcasting is a cast
- The exact type of the object is lost
- What gets printed in the CellPhone example?
- **Tune.play()** (what were you expecting?)
- Is this “the right thing to do”?
- The alternative: write separate **ring()** methods for each subclass of Tune?

Another Example

```
class CellPhone {
    CellPhone() { //…}
    public void ring(Tune t) { t.play(); }
}
class Tune {
    Tune() { //…}
    public void play() {
        System.out.println("Tune.play()");
    }
}
class ObnoxiousTune extends Tune{
    ObnoxiousTune() { // …}
    public void play() {
        System.out.println("ObnoxiousTune.play()");
    }
}
```

Polymorphism

- The second example prints **ObnoxiousTune.play()**
- Since an **ObnoxiousTune** object was sent to **ring()**, the **ObnoxiousTune**'s **play()** method is called.
- This is called "polymorphism"
- Most people think it's "the right thing to do"

Method-Call Binding

- The correct method is attached ("bound") to the call at runtime.
- The runtime system discovers the type of object sent to **ring()**, and then selects the appropriate **play()** method:
  - if it's a **Tune** object, **Tune**'s **play()** is called
  - if it's an **ObnoxiousTune** object, **ObnoxiousTune**'s **play()** is called

Is Java *Always* Late-Binding?

- Yes, always, except for **final, static and private** methods.
- **final** methods can't be overridden, so the compiler knows to bind it.
- The compiler *may* be able to do some speed optimizations for a final method, but we programmers are usually lousy at estimating where speed bottlenecks are…
- In C++, binding is *always* at compile-time, unless the programmer says otherwise.

Another Polymorphism Example

```
public static void main(String[] args) {
    CellPhone noiseMaker = new CellPhone();
    Tune t;
    double d = Math.random();
    if (d > 0.5)
        t = new Tune();
    else
        t = new ObnoxiousTune();
    noiseMaker.ring(t);
}
```

References and Subclasses

- We’ve seen that an **ObnoxiousTune** object can be supplied where a **Tune** object is expected.
- Similarly, a **Tune** reference can refer to an **ObnoxiousTune** object.
- In both cases, the basic question is: “Is an **ObnoxiousTune** really a proper **Tune**?” Yes!
- This doesn’t work the other way around: any old **Tune** *may not be* an **ObnoxiousTune**.
- So, an **ObnoxiousTune** reference cannot refer to a **Tune** object.

Let's Fool the Compiler!

```
public static void main(String[] args) {
    CellPhone noiseMaker = new CellPhone();
    Tune t1 = new Tune();
    Tune t2 = new ObnoxiousTune();
    noiseMaker.ring(t1);
    noiseMaker.ring((Tune)t2);
}
```

Nothing changes… **ObnoxiousTune.play()** is still printed.

Let's Fool the Compiler!

```
public static void main(String[] args) {
    CellPhone noiseMaker = new CellPhone();
    Tune t1 = new Tune();
    Tune t2 = new ObnoxiousTune();
    noiseMaker.ring(t1);
    noiseMaker.ring((ObnoxiousTune) t2);
}
```

Nothing changes…

Let’s Fool the Compiler!

```
public static void main(String[] args) {
    CellPhone noiseMaker = new CellPhone();
    Tune t1 = new Tune();
    Tune t2 = new ObnoxiousTune();
    noiseMaker.ring((ObnoxiousTune) t1);
    noiseMaker.ring(t2);
}
```

This compiles, but gets a **CastClassException** at runtime (even though **Tune** has a **play()** method). t1 has been casted from Tune to ObnoxiousTune Parent to Child. “Downcasting” can be dangerous!

Extensibility I

```
public static void main(String[] args) {
    CellPhone noiseMaker = new CellPhone();
    SimpleInput keyboard = new SimpleInput();
    System.out.println("Enter number of tunes:");
    int numTunes = keyboard.nextInt();
    Tune[] tunes = new Tune[numTunes];
    for (int i = 0; i < numTunes; i++) {
       System.out.println("Enter tune type");
       System.out.println("(Tune=1, ObnoxiousTune=2)");
       int tuneType = keyboard.nextInt();
          switch(tuneType) {
             case 1:  tunes[i] = new Tune(); break;
             case 2:  tunes[i] = new ObnoxiousTune(); break;
             default: tunes[i] = new Tune(); break;
          }
    }
    for (int i = 0; i < tunes.length; i++)
          noiseMaker.ring(tunes[i]);
}
```

Extensibility II

```
public class NiceTune extends Tune {
    NiceTune() {}
    public void play() {
        System.out.println("NiceTune.play()");
    }
}
```

Extensibilty III

```
public static void main(String[] args) {
    CellPhone noiseMaker = new CellPhone();
    SimpleInput keyboard = new SimpleInput();
    System.out.println("Enter number of tunes:");
    int numTunes = keyboard.nextInt();
    Tune[] tunes = new Tune[numTunes];
    for (int i = 0; i < numTunes; i++) {
      System.out.println("Enter tune type");
      System.out.println("(Tune=1, ObnoxiousTune=2, NiceTune = 3)");
      int tuneType = keyboard.nextInt();
      switch(tuneType) {
            case 1:  tunes[i] = new Tune(); break;
            case 2:  tunes[i] = new ObnoxiousTune(); break;
            case 3:  tunes[i] = new NiceTune(); break;//extends Tune()
            default: tunes[i] = new Tune(); break;
      }
    }
    for (int i = 0; i < tunes.length; i++)
            noiseMaker.ring(tunes[i]);
}
```

Another Example: Arithmetic

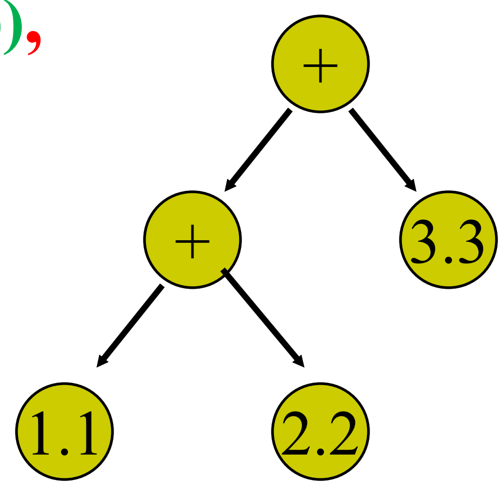
```
public class Node {
    public Node() {}
    public double eval() {
        System.out.println("Error: eval Node");
        return 0;
    }
}
public class Binop extends Node {
    protected Node lChild, rChild;
    public Binop(Node l, Node r) {
            lChild = l;
            rChild = r;
    }
}
```

Arithmetic (cont.)

```
public class Plus extends Binop {
    public Plus(Node l, Node r) {
        super(l, r);// l, r of Binop
    }
    public double eval() {
        return lChild.eval() + rChild.eval();
        //protected Note can
        //Accessed by subclass
   }
}
public class Const extends Node {
    private double value;
    public Const(double d) { value = d; }
    public double eval() { return value; }
}
```

Arithmetic (cont.)

```
public class TestArithmetic {
    // evaluate 1.1 + 2.2 + 3.3
    public static void main(String[] args) {
        Node n = new Plus(
            new Plus(
            new Const(1.1), new Const(2.2)),
            new Const(3.3));
        System.out.println(""+ n.eval());
    }
}
```

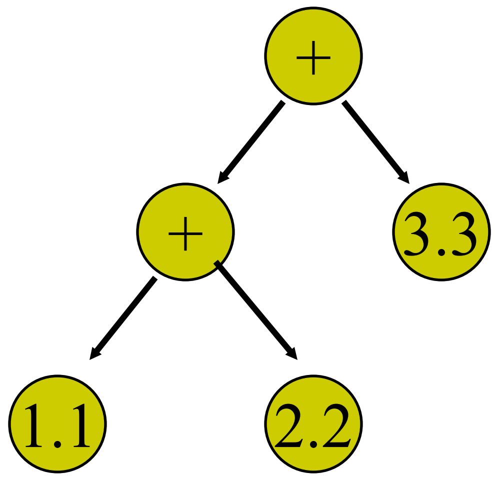


Arithmetic (cont.)

```
Node n1 = new Const(1.1);
Node n2 = new Const(2.2);
Node n3 = new Plus(n1, n2);
Node n4 = new Const(3.3);
Node n5 = new Plus(n3, n4);
```

Arithmetic (cont.)

- Binary operators create a binary tree.
- Constants are “leaf” nodes.
- We could easily add more operators and terminals.



Extensibility

- Both the Tunes and Arithmetic programs allow additional subclasses to be constructed and easily integrated.
- Polymorphism is the key in letting this happen.
- It was OK to make Tune objects, but we should never make Node objects.
- How to prevent this?

Abstract Classes

- Can not be used to create objects
- To access it, must be call via a class inherited from the abstract class

Abstract Classes

- As always, get the compiler involved in enforcing our decisions.

```
public abstract class Node {
    public Node() {}
    public abstract double eval();
}
```

- Now it's a compiler error if we try to make a Node object (but Node references are OK).

  Node n = new Node(); //**wrong, compiler error**

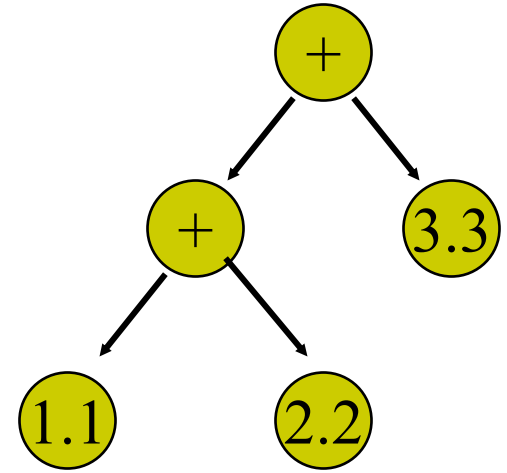
- Subclasses are abstract (and we must so state) until all abstract methods have been defined.

Order of Construction

- Put print statements into the constructors in the Arithmetic example. The output is:

Node constructor Node n1
1. Const constructor 1.1
n1 = new Const(1.1)
Node constructor Node n2
2. Const constructor 2.2
n2 = new Const(2.2)
Node constructor Node n3
Binop constructor
3. Plus constructor
n3 = new Plus (n1, n2)
Node constructor Node n4
4. Const constructor 3.3
n4= new Const(3.3)
Node constructor Node n5
Binop constructor
5. Plus constructor 6.6
n5 = new Plus (n4, n5)

```
Node n = new Plus(
        new Plus(
        new Const(1.1), new Const(2.2)),
        new Const(3.3));
```



Construction: Glyph Example

```
abstract class Glyph {
    abstract void draw();
    Glyph() {
        System.out.println("Glyph() before draw");
        draw();
        System.out.println("Glyph() after draw");
    }
}
```

Glyph Example (cont.)

```
class RoundGlyph extends Glyph {
    int radius = 1;
    RoundGlyph(int r) {
        radius = r;
        System.out.println("RoundGlyph(), radius=" + radius);
    }
    void draw() {
        System.out.println("RoundGlyph.draw(), radius=" + radius);
    }

public class GlyphTest {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }
}
```

Glyph Example (cont.)

- This produces

  Glyph() before draw()
  RoundGlyph.draw(), radius=0
  Glyph() after draw()
  RoundGlyph(), radius= 5

- Guideline for constructors:
  - "Do as little as possible to set the object into a good state, and if you can possibly avoid it, don't call any methods."

But What If **Draw()** Isn't Abstract?

```
abstract class Glyph {
    void draw() { System.out.println("Glyph.draw()"); }
    abstract void doNothing();  // added to keep Glyph abstract
    Glyph() {
        System.out.println("Glyph() before draw");
        draw();
        System.out.println("Glyph() after draw");
    }
}
```

Glyph Example (cont.)

- The **Glyph** constructor is at work creating the **Glyph** “sub-object” within a **RoundGlyph** object.
- **draw()** is overridden in **RoundGlyph**.
- A **RoundGlyph** object is being created.
- **RoundGlyph**’s **draw()** method is called
- Polymorphism rules!

Inheritance is Cool, But…
Composition + Inheritance is Cooler

```
abstract class Actor {
    abstract void act();
}
class HappyActor extends Actor {
    public void act() { //…}
}
class SadActor extends Actor {
    public void act() { //…}
}
class Stage {
    Actor a = new HappyActor();
    void change() { a = new SadActor(); }
    void go() { a.act(); }
}
```

```
public class Transmogrify {
    public static void
        main(String[] args){
        Stage s = new Stage();
        s.go();   //happy actor
        s.change();
        s.go()   // sad actor
    }
}
```

Tips for Inheritance

- Place common operations and fields in the superclass.
- Don’t use protected fields very often.
- Use inheritance to model the “is-a” relationship.
- Don’t use inheritance unless all inherited methods make sense.
- Use polymorphism, not type information.
- Don’t overuse reflection.

Digression: The **Object** Class

- **Object** is the ancestor of *every* class.
- We don't need to say, e.g.,

  class Employee extends Object

- We can use an **Object** reference to refer to any object.

  Object obj = new Employee("Harry Hacker", 35000);

- But to use any of the methods of Employee, we must cast to the correct type.

  int salary = ((Employee) obj).getSalary();

Some **Object** Methods

- **clone()**: Creates and returns a copy of this object. Returns an **Object**.
- **equals()**: Indicates whether some other object is "equal to" this one. Returns a **Boolean**.
- **getClass()**: Returns the runtime class of an object. Returns a **Class**.
- **toString()**: Returns a string representation of the object. Returns a **String**.

The **equals()** Method

- As implemented in the **Object** class, this just tests whether two references point to the same memory.
- This isn’t usually what we want.
- Override to get correct behavior.
- Pay attention to the Java language spec. to do it right.

The Rules For **equals()**

- Reflexive: **x.equals(x)** is **true**
- Symmetric: if **x.equals(y)** then **y.equals(x)**
- Transitive: if **x.equals(y)** and **y.equals(z)** then **x.equals(z)**
- Consistent: if **x.equals(y)** now, and **x** and **y** don’t change, then **x.equals(y)** later
- If **x** is non-null, **x.equals(null)** is **false**

equals() Example

```
class Employee {
    private String name;
    private double salary;
    private Date hireDay;
    public boolean equals(Object otherObject) {
        if (this == otherObject) return true;
        if (otherObject == null) return false;
        if (getClass() != otherObject.getClass())
            return false;
        Employee other = (Employee)otherObject;
        return name.equals(other.name)
           && salary == other.salary
           && hireDay.equals(other.hireDay);
    }
}
```

Digression: Object Wrappers

- Sometimes you want to put a number where an object is required.

```
ArrayList list = new ArrayList();
//! list.add(3.14);
```

- But you can wrap the number in a Double:

```
list.add(new Double(3.14));
```

- Later, you can extract the number:

```
double x = ((Double)list.get(n)).doubleValue();
```

- There are **Integer**, **Long**, **Float**, **Double**, **Short**, **Byte**, **Character**, **Void** and **Boolean**

Interpreting **Strings** as Numbers

- To get an integer from the **String** s, use

```
int x = Integer.parseInt(s);
```

- **parseInt()** is a static method in **Integer**.
- Another way:

```
NumberFormat formatter =
        NumberFormat.getNumberInstance();
Number n = formatter.parse(s);
```

- Number is an abstract class, and n is either a Double or a Long

```
if (n instanceOf Double) Double d = (Double)n;
```

Digression: The **Class** Class

- We saw that polymorphism is accomplished by the Java runtime system, which keeps track of every object's type.
- We can get the same information in code:

```
Employee e;
…
Class cl = e.getClass();
System.out.println(cl.getName() + " " + e.getName());
```

- Recall **e** can refer to an **Employee** or a **Manager**.

Other Ways to Get a **Class**

- With the **forName()** method:

```
String className = "Manager";
// the following line may throw a checked exception
Class cl = Class.forName(className);
```

- or, more directly:

```
Class c11 = Manager.class;
Class c12 = int.class;
Class c13 = Double.class;
```

Digression in a Digression

- A “checked exception” needs to be handled.

```
try {
    String className = …;  // get class name
    // the following line may throw a checked exception
    Class cl = Class.forName(className);
    … // do something with cl
}
catch(Exception e) {
    e.printStackTrace();
}
```

Methods of the **Class** Class

- The concept of “reflection” is supported by **Class** methods discover superclass info, constructors, methods and fields.

```
import java.lang.reflect.*;  // defines Constructor, Field, Method
…
String className  = "Manager";
Class cl = Class.forName(className);
Field[] fields = cl.getDeclaredFields();
for (int i = 0; i < fields.length; i++)
   System.out.println(fields[i].getName());
```

“Growing” an Array

```
static Object arrayGrow(Object a) {
    Class cl = a.getClass();
    if (!cl.isArray()) return null;
    Class componentType = cl.getComponentType();
    int length = Array.getLength(a);
    int newLength = length * 11 / 10 + 10;
    Object newArray = Array.newInstance (componentType,
        newLength);
    System.arraycopy(a, 0, newArray, 0, length);
    return newArray;
}
```