Polymorphism

Slides adapted from Prof. Steven Roehrig and Sakir YUCEL (CMU)

- 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 StrangeTune extends Tune{
    StrangeTune() {
        System.out.println("obnoxious.play()");
    }
}
```

An **StrangeTune** “is-a” **Tune**

```
class CellPhone {
    cellPhone() { //...}
    public void ring(Tune t) { t.play(); }
}
class Tune {
    public void play() {System.out.println("Tune.play()");     }
}
class StrangeTune extends Tune{
    StrangeTune() { System.out.println("obnoxious.play()"); }
}
public class DisruptLecture {
    public static void main(String[] args) {
        CellPhone noiseMaker = new CellPhone();
        StrangeTune ot = new StrangeTune();
        noiseMaker.ring(ot);  // ot works though Tune called for
    }
}
```

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

Polymorphism

- The second example prints **StrangeTune.play()**
- Since an **StrangeTune** object was sent to **ring()**, the **StrangeTune**'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 **StrangeTune** object, **StrangeTune**'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 StrangeTune();
        noiseMaker.ring(t);
}
```

References and Subclasses

- We've seen that an **StrangeTune** object can be supplied where a **Tune** object is expected.
- Similarly, a **Tune** reference can refer to an **StrangeTune** object.
- In both cases, the basic question is: "Is an **StrangeTune** really a proper **Tune**?" Yes!
- This doesn't work the other way around: any old **Tune** *may not be* an **StrangeTune**.
- So, an **StrangeTune** 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 StrangeTune();
        noiseMaker.ring(t1);
        noiseMaker.ring((Tune)t2);
}
```

Nothing changes… **StrangeTune.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 StrangeTune();
        noiseMaker.ring(t1);
        noiseMaker.ring((StrangeTune) t2);
}
```

Let's Fool the Compiler!

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

This compiles, but gets a **CastClassException** at runtime (even though **Tune** has a **play()** method).

"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, StrangeTune=2)");
            int tuneType = keyboard.nextInt();
           switch(tuneType) {
               case 1:  tunes[i] = new Tune(); break;
               case 2:  tunes[i] = new StrangeTune(); break;
                   default: tunes[i] = new Tune(); break;
           }
        }
```

Extensibility II

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

Extensibilty III

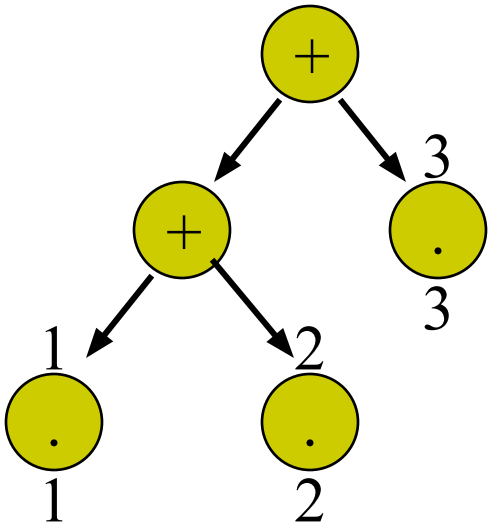
```
public static void main(String[] args) {
        CellPhone noiseMaker = new CellPhone();
        SimpleInput keyboard = new SimpleInput();
        System.out.println("Enter number of tunes:");
        int numTunes = keyboard.nextInt();
        Tune[] tunes = new Tune[numTunes];
        for (int i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

Another example: Arithmetic

```
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);
```

Another example: Arithmetic

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



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;
    }
}
```

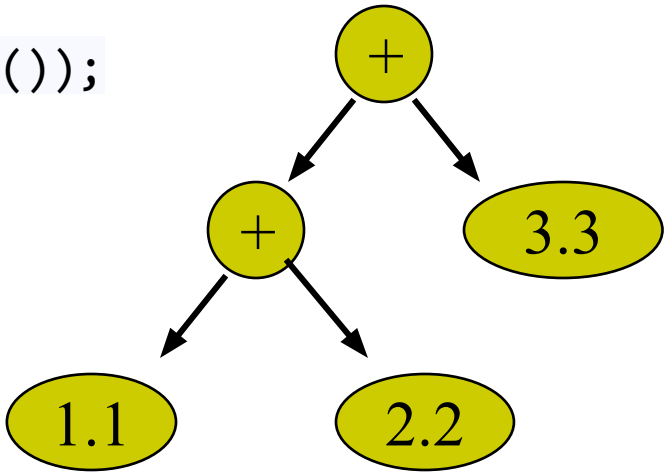
Arithmetic (cont.)

```
public class 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;
    }
}
```

```
public class Plus extends Binop
    public Plus(Node l, Node r)
        super(l, r);
    }
    public double eval() {
        return lChild.eval() +
rChild.eval();
    }
}
public class Const extends Node
    private double value;
    public Const(double d) { va
d; }
    public double eval() { retu
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());
    }
}
```



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

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

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

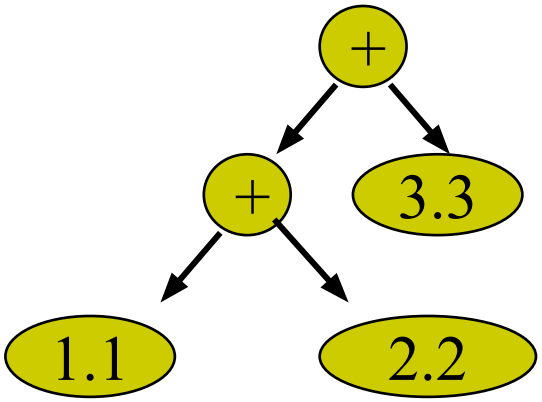
- Now it's a compiler error if we try to make a Node object (but Node references are OK).
- Subclasses are abstract (and we must so state) until all abstract methods have been defined.

Order of Construction

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

Node constructor
Const constructor 1.1
Node constructor
Const constructor 2.2
Node constructor
Binop constructor
Plus constructor
Node constructor
Const constructor 3.3
Node constructor
Binop constructor
Plus constructor
6.6

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



Construction: Kabartma Example

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

Kabartma Example (cont.)

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

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

Kabartma Example (cont.)

- This produces

```
Kabartma() before draw()
YuvarlakKabartma.draw(), radius=0
Kabartma() after draw()
YuvarlakKabartma(), 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 Kabartma {
    void draw() { System.out.println("Kabartma.draw()"); }
    abstract void doNothing();  // added to keep Kabartma
abstract
    Kabartma() {
        System.out.println("Kabartma() before draw");
        draw();
        System.out.println("Kabartma() after draw");
    }
}
```

Kabartma Example (cont.)

- The **Kabartma** constructor is at work creating the **Kabartma** "sub-object" within a **YuvarlakKabartma** object.
- **draw()** is overridden in **YuvarlakKabartma**.
- A **YuvarlakKabartma** object is being created.
- **YuvarlakKabartma**'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 Convert {
  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.

Reflection example

```
// import java.lang.reflect.Method;
import java.lang.reflect.Field;
//import java.lang.reflect.Constructor;
public class KabartmaTest {
    public static void main(String[] args) {
        YuvarlakKabartma yuv = new YuvarlakKabartma(5);
        System.out.println(yuv.getClass().getName());
        Class cls = yuv.getClass();
        try {
            Field field = cls.getDeclaredField("pX");
            field.setAccessible(true);
            field.set(yuv,10);
            System.out.println("Field XX " + field.get(yuv));
        }catch (Exception e){
         ;
        }
```

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**

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**.

Methods of the **Class** Class

- The concept of "reflection" is supported by **Class** methods that let you 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());
```