More About Inheritance

COMP 110
Summer II 2012

Peter Lincoln
7/16/2012
Outline

- Review Inheritance

- Sports Example
  - Dynamic Binding

- The Object Class
  - toString & equals
Review: Inheritance

- What is a base class?

- What is a derived class?
Review: The *is-a* Relationship

- A Doctoral student is a Grad student
- A Grad student is a person
- A Student is a Person

Is a Person a Student?
- Not necessarily!
Review: Overrriding Methods

- A method in a derived class with the same signature (name & arguments) as a method in a base class replaces it for all objects of the derived class.
Overriding Methods: Why is it useful?

- You often want derived classes to perform custom behavior
  - Displaying custom information
  - Drawing custom shapes
  - React to input differently
Let’s try another example...
Type compatibilities

- Given this inheritance hierarchy...

```
Person
   Athlete
     HighJumper
     ExtremeAthlete
       Skydiver
       XGamesSkater
```
Is this code legal?

- Person p = new Person();
- Yes!

- Person
  - Athlete
    - HighJumper
    - ExtremeAthlete
      - Skydiver
      - XGamesSkater
Is this code legal?

- HighJumper h = new HighJumper();
- Yes!
Is this code legal?

- Person p = new Athlete();
  - Yes! An Athlete is a Person, so this is okay

![Class Diagram]

Person
  ↓
Athlete
  ↓
HighJumper
  ↓
ExtremeAthlete
  ↓
Skydiver
  ↓
XGamesSkater
Skydiver s = new Person();

No! A Person **is not necessarily** a Skydiver, so this is illegal
Is this code legal?

- Athlete ath = new Athlete();
  XGamesSkater xgs = ath;

- No! An Athlete **is not necessarily** an XGamesSkater, so this is illegal
More on the Sports Example

- Suppose every class had a `jump()` method with a different action

<table>
<thead>
<tr>
<th>Class</th>
<th>Printed Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>“Whee!”</td>
</tr>
<tr>
<td>Athlete</td>
<td>“I jump well!”</td>
</tr>
<tr>
<td>HighJumper</td>
<td>“I jump REALLY HIGH!”</td>
</tr>
<tr>
<td>ShotPutter</td>
<td>“I don’t jump very much”</td>
</tr>
<tr>
<td>ExtremeAthlete</td>
<td>“EXTREMEEEEEEEEE JUMP!”</td>
</tr>
<tr>
<td>XGamesSkater</td>
<td>“360 nollie to frontside air”</td>
</tr>
</tbody>
</table>
What would be displayed?

- Person per = new Person();
  per.jump();
  “Whee!”

<table>
<thead>
<tr>
<th>Class</th>
<th>Printed Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>“Whee!”</td>
</tr>
<tr>
<td>Athlete</td>
<td>“I jump well!”</td>
</tr>
<tr>
<td>HighJumper</td>
<td>“I jump REALLY HIGH!”</td>
</tr>
<tr>
<td>ShotPutter</td>
<td>“I don’t jump very much”</td>
</tr>
<tr>
<td>ExtremeAthlete</td>
<td>“EXTREMEEEEEE JUMP!”</td>
</tr>
<tr>
<td>XGamesSkater</td>
<td>“360 nollie to frontside air”</td>
</tr>
</tbody>
</table>
What would be displayed?

- `ExtremeAthlete ea = new ExtremeAthlete();
  ea.jump();`
- “EXTREMEEEEEEEEE JUMP!”

<table>
<thead>
<tr>
<th>Class</th>
<th>Printed Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>“Whee!”</td>
</tr>
<tr>
<td>Athlete</td>
<td>“I jump well!”</td>
</tr>
<tr>
<td>HighJumper</td>
<td>“I jump REALLY HIGH!”</td>
</tr>
<tr>
<td>ShotPutter</td>
<td>“I don’t jump very much”</td>
</tr>
<tr>
<td>ExtremeAthlete</td>
<td>“EXTREMEEEEEEE JUMP!”</td>
</tr>
<tr>
<td>XGamesSkater</td>
<td>“360 nollie to frontside air”</td>
</tr>
</tbody>
</table>
What would be displayed?

- Athlete ath = new XGamesSkater();
  ath.jump();
  “360 nollie to frontside air”

<table>
<thead>
<tr>
<th>Class</th>
<th>Printed Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>“Whee!”</td>
</tr>
<tr>
<td>Athlete</td>
<td>“I jump well!”</td>
</tr>
<tr>
<td>HighJumper</td>
<td>“I jump REALLY HIGH!”</td>
</tr>
<tr>
<td>ShotPutter</td>
<td>“I don’t jump very much”</td>
</tr>
<tr>
<td>ExtremeAthlete</td>
<td>“EXTREMEEEEEEE JUMP!”</td>
</tr>
<tr>
<td>XGamesSkater</td>
<td>“360 nollie to frontside air”</td>
</tr>
</tbody>
</table>
What’s going on?

- We are creeping up to the idea of polymorphism
  - Enables the substitution of one object for another as long as the object’s have the same interface
public static void jump3Times(Person p)
{
    p.jump();
    p.jump();
    p.jump();
}

public static void main(String[] args)
{
    XGamesSkater xgs = new XGamesSkater();
    Athlete ath = new Athlete();
    jump3Times(xgs);
    jump3Times(ath);
}
What if we wrote a new class?

- Note that we wrote the class Person before any of the derived classes were written.

- We can create a new class that inherits from Person, and the correct jump method will be called because of *dynamic binding*.
Dynamic Binding

- The method invocation is not bound to the method definition until the program executes

```java
public class SkiJumper extends ExtremeAthlete {
    public void jump() {
        System.out.println("Launch off a ramp and land on snow");
    }
}

public static void main(String[] args) {
    SkiJumper sj = new SkiJumper();
    jump3Times(sj);
}
```
The Object Class

- Every class in Java is derived from the class Object
  - Every class in Java is an Object
The Object Class

- Object has several public methods that are inherited by subclasses

- Two commonly override Object methods
  - toString
  - equals
Calling System.out.println() on an object

There is a version of System.out.println that takes an Object as a parameter. What happens if we do this?

- Person p = new Person();
  System.out.println(p);

We get something like:

Person@addbf1

The class name @ hash code
The `toString` method

- Every class has a `toString` method, inherited from `Object`

```java
public String toString()
```

- Intent is that `toString` be overridden, so subclasses can return a custom string representation
When we call `System.out.println()` on an object...

1. The object’s `toString` method is called
2. The String that is returned by the `toString` method is printed

```java
public class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public String toString() {
        return "Name: " + name;
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        Person per = new Person("Apu");
        System.out.println(per);
    }
}
```

Output:

```
Person@addbf1
Name: Apu
```
What if we have a derived class?

(Assume the Person class has a getName method)

```java
public class Student extends Person {
    private int id;
    public Student(String name, int id) {
        super(name);
        this.id = id;
    }
    public String toString() {
        return "Name: " + getName() + ", ID: " + id;
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        Student std = new Student("Apu", 17832);
        System.out.println(std);
    }
}
```

Output:
Name: Apu, ID: 17832
What if we have a derived class?

Would this compile?

```java
public class Test {
    public static void main(String[] args) {
        Person per = new Student("Apu", 17832);
        System.out.println(per);
    }
}
```

Yes. What is the output?

- Name: Apu, ID: 17832

Automatically calls Student’s toString method because per is of type Student
The equals method

- The equals method performs equality testing

- First try:
  - `public boolean equals(Student std) {
    return (this.id == std.id);
  }

- However, we really want to be able to test if two Objects are equal
The equals method

- Object has an equals method
  - Subclasses should override it

- Default Object equals method:
  ```java
  public boolean equals(Object obj)
  {
    return (this == obj);
  }
  ```

- What does this method do?
  - Returns whether this has the same address as obj
  - This is the default behavior for subclasses
Second try:

```java
public boolean equals(Object obj) {
    Student otherStudent = (Student) obj;
    return (this.id == otherStudent.id);
}
```

What does this method do?

- Typecasts the incoming Object to a Student
- Returns whether this has the same id as otherStudent
The equals method

```java
public boolean equals(Object obj) {
    Student otherStudent = (Student)obj;
    return (this.id == otherStudent.id);
}
```

- Why do we need to typecast?
  - Object does not have an id, obj.id would not compile

- What’s the problem with this method?
  - What if the object passed in is not actually a Student?
  - The typecast will fail and we will get a runtime error
The `instanceof` operator

We can test whether an object is of a certain class type:

```java
if (obj instanceof Student) {
    System.out.println("obj is an instance" + "of the class Student");
}
```

Syntax:

- `objectName instanceof ClassName`

Use this operator in the `equals` method
The equals method

- Third try

```java
public boolean equals(Object obj)
{
    if ((obj != null) && (obj instanceof Student))
    {
        Student otherStudent = (Student)obj;
        return (this.id == otherStudent.id);
    }
    return false;
}
```

- Reminder: null is a special constant that can be assigned to a variable of a class type – means that the variable does not refer to anything right now
Questions?
Logistics

- Next:
  - Lab 7