COMP 110
Inheritance and polymorphism

Darrell Bethea
June 8, 2011
Announcements

- Program 4 due Friday
- Final exam
  - Comprehensive
  - Monday, 6/13, 8–11 AM
  - SN014
Questions?
Today in COMP 110

- Inheritance and polymorphism
Person has a jump method, so all subclasses have a jump method
Each subclass has its own jump functionality

public class Person {
    public void jump() {
        System.out.println("Whee!");
    }
}

public class Athlete extends Person {
    public void jump() {
        System.out.println("I jump really well!");
    }
}
Review: type compatibilities

- ExtremeAthlete is an Athlete
- XGamesSkater is a Person
- Person is not necessarily a Skydiver
Review: type compatibilities

- Person p = new ExtremeAthlete();
  - legal
- Athlete a = new Athlete();
  - legal
- XGamesSkater xgs = new Person();
  - illegal
Polymorphism

- “many forms”
- Enables the substitution of one object for another as long as the objects 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.
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);
}
```
Every class in Java is derived from the class Object
- Every class in Java is an Object
Object has several public methods that are inherited by subclasses

Two commonly overridden Object methods:
- toString
- equals
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

    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...

- the object’s `toString` method is called
- 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;
    }
}

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

Output:

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
Would this compile?
Yes. What is the output?
Automatically calls Student’s toString method because p is of type Student

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

Output:
Name: Apu, ID: 17832
The equals method

- First try:

```java
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

```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
The equals method

- 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`
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:

  ```java
  object instanceof Class_Name
  ```

- 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.