COMP 110-001
Inheritance and Polymorphism

Yi Hong

June 09, 2015

Today

- Inheritance and polymorphism

Inheritance and Polymorphism

- *Inheritance* allows you to define a base class and derive classes from the base class
- *Polymorphism* allows you to make changes in the method definition for the derived classes and have those changes apply to the methods written in the base class → "Many forms"

Calling a Derived Class' Overridden Method

```
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

```
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);
}
```

Another Example of Polymorphism

```
public class PolymorphismDemo
{
    public static void main(String[] args)
    {
        Person[] people = new Person[4];
        people[0] = new Undergraduate("Cotty, Manny", 4910, 1);
        people[1] = new Undergraduate("Kick, Anita", 9931, 2);
        people[2] = new Student("DeBanque, Robin", 8812);
        people[3] = new Undergraduate("Bugg, June", 9901, 4);
        for (Person p : people)
        {
            p.writeOutput();
            System.out.println();
        }
    }
}
```

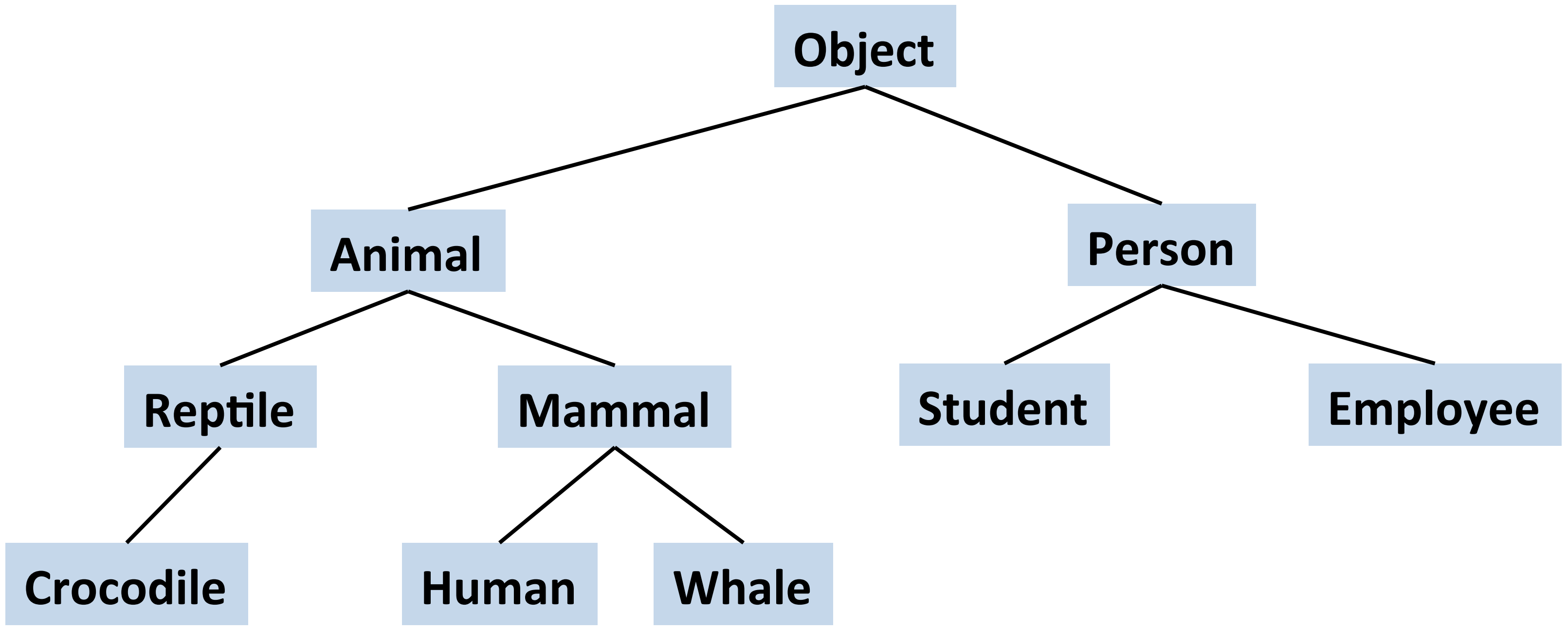
Even though `p` *is of type* `Person`*, the* `writeOutput` *method associated with* `Undergraduate` *or* `Student` *is invoked depending upon which class was used to create the object.*

Dynamic Binding and Polymorphism

- Dynamic binding: the method is not bound to an invocation of the method until run time when the method called
- Polymorphism: associate many meanings to one method name through the dynamic binding mechanism

The Class Object

- Every class in Java is derived from the class Object
    - Every class in Java *is an* Object



The class Object

- Object has several public methods that are inherited by subclasses
- Two commonly overridden Object methods:
  - `toString:`
    - takes no arguments, and returns all the data in an object, packaged into a string
  - `equals`
    - Compares two objects

Calling System.out.println()

- 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;
    }
}
```

```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)

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;
    }
}

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?

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

- Yes. What is the output?

Output:

```
Name: Apu, ID: 17832
```

- Automatically calls Student's toString method because *per* is of type Student

The `equals` method

- 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

```
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

```
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

```
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

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

- Syntax:

```
object instanceof Class_Name
```

- Use this operator in the equals method

The `equals` method

- Third try

```
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

Next Class

- Exception handling
- File I/O