COMP 110-001 Inheritance Basics

Yi Hong

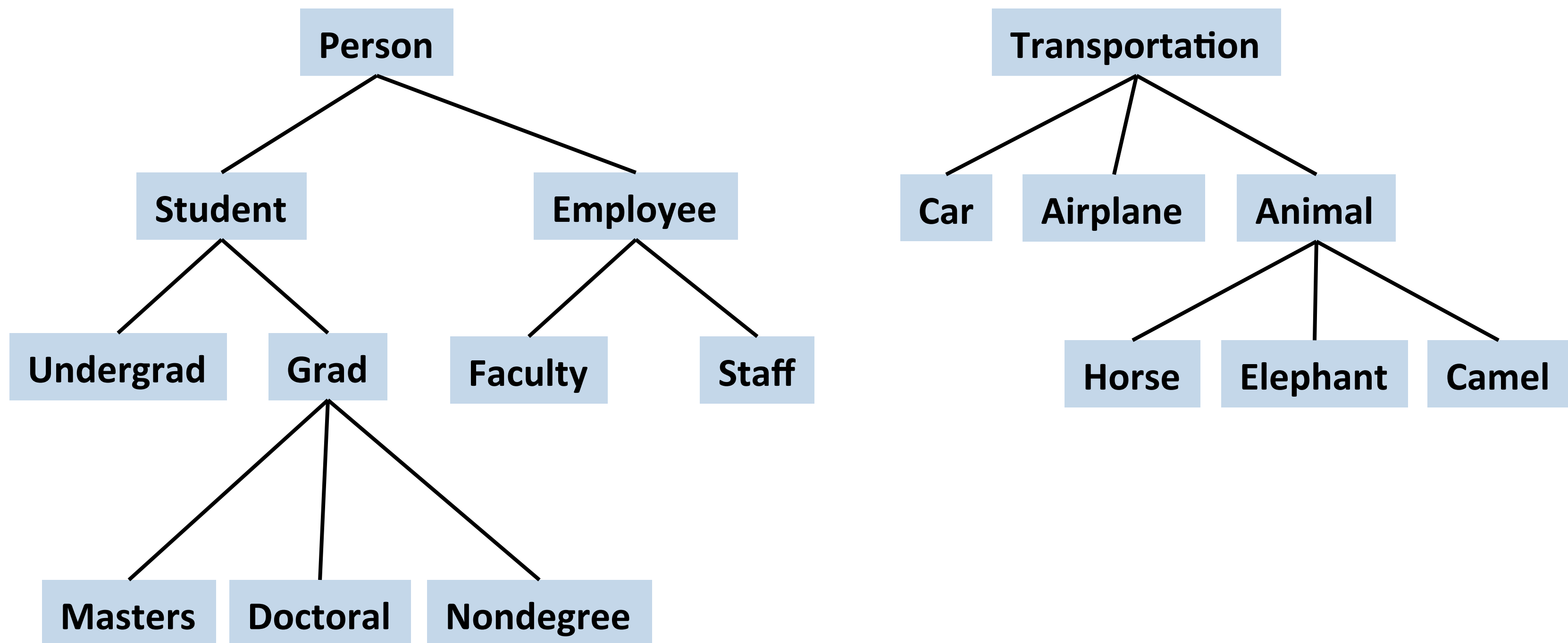
June 09, 2015

Today

- Inheritance

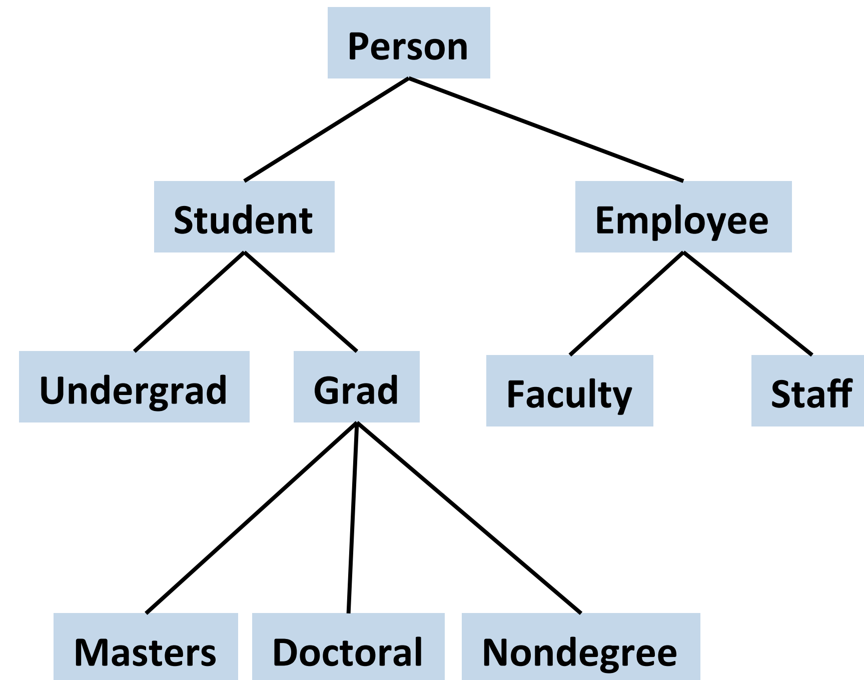
Inheritance

- We have discussed before how classes of objects can have relationships



Inheritance

- Define a general class
- Later, define specialized classes based on the general class
- These specialized classes *inherit* properties from the general class

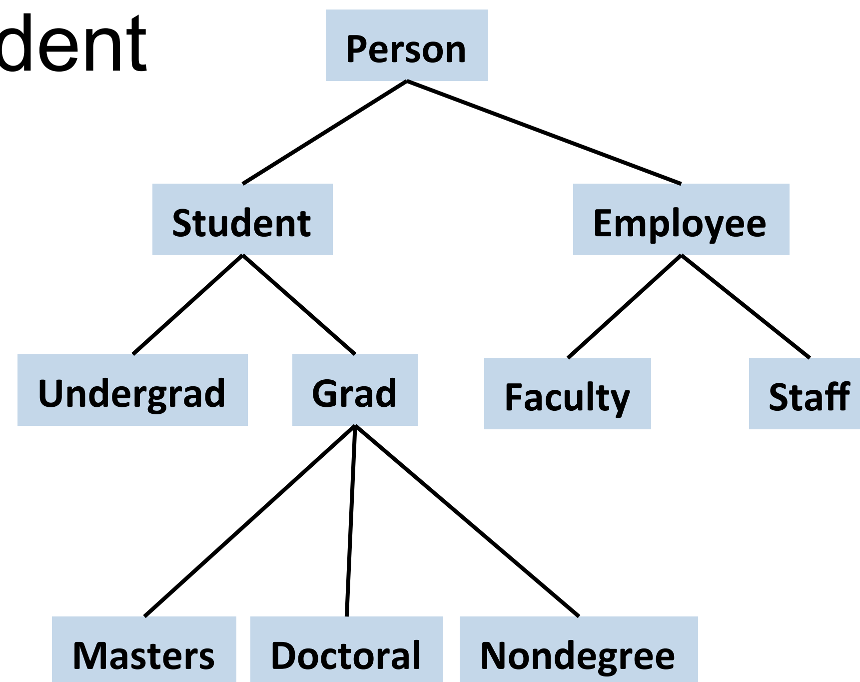


Inheritance

- What are some properties of a Person?
  - Name, height, weight, age
- How about a Student?
  - ID, major
- Does a Student have a name, height, weight, and age?
  - Student *inherits* these properties from Person

The *is-a* Relationship

- This inheritance relationship is known as an *is-a relationship*
- A Doctoral student *is a* Grad student
- A Grad student *is a* Student
- A Student *is a* Person
- Is a Person a Student?
  - Not necessarily!



Base Class

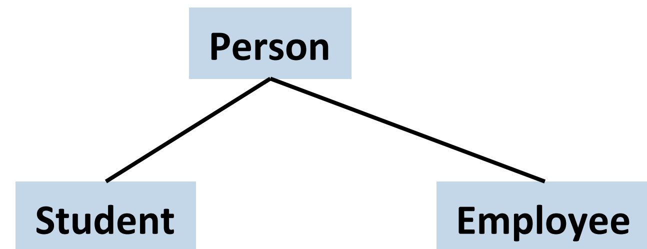
- Our general class is called a ***base class***
  - Also called a ***parent class*** or a ***superclass***

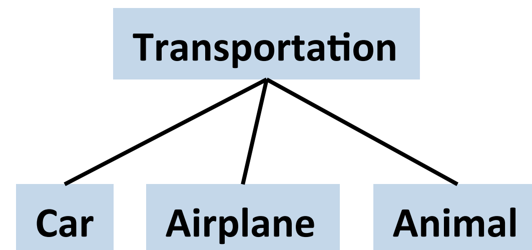
- Examples:
  - Person, Transportation

Derived Class

- A specialized class that inherits properties from a base class is called a ***derived class***
  - Also called a ***child class*** or a ***subclass***

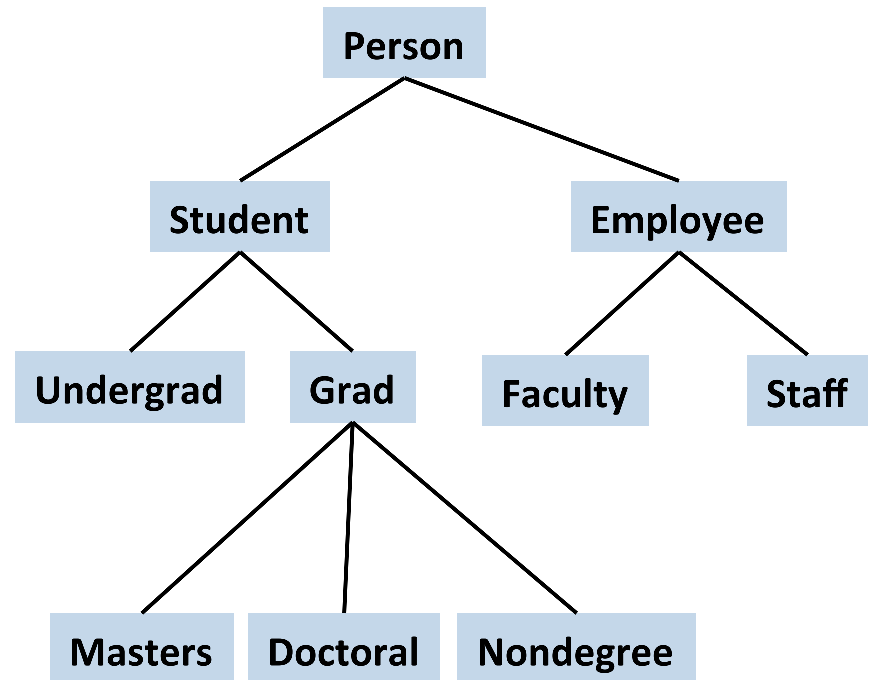
- Examples:
  - Student ***is-a*** Person
  - Employee ***is-a*** Person
  - Car ***is-a*** form of Transportation
  - Animal ***is-a*** form of Transportation





Child (Derived) Classes Can Be Parent (Base) Classes

- Student is a child class of Person
- Student is also the parent class of Undergrad and Grad



Why Is Inheritance Useful?

- Enables you to define shared properties and actions *once*
- Derived classes can perform the same actions as base classes without having to redefine the actions
  - If desired, the actions *can* be redefined – more on this later

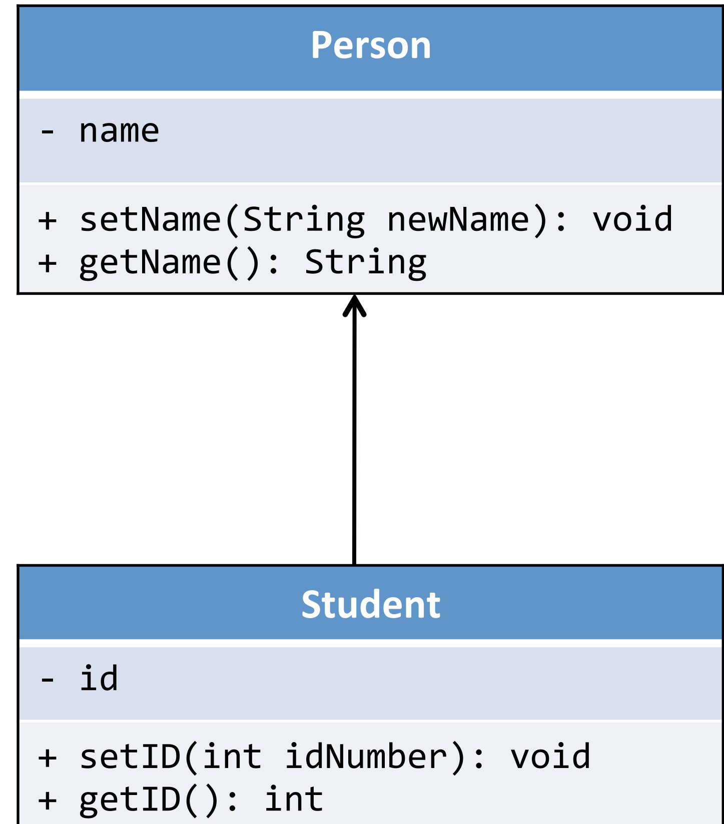
How Does This Work in Java?

```
public class Person
{
    private String name;
    public Person()
    {
        name = “No name yet”;
    }
    public void setName(String newName)
    {
        name = newName;
    }
    public String getName()
    {
        return name;
    }
}
```

Person
- name
+ setName(String newName): void + getName(): String

How Does This Work in Java?

```java
public class Student extends Person
{
    private int id;
    public Student()
    {
        super();
        id = 0;
    }
    public Student(String stdName, int idNumber)
    {
        setName(stdName);
        setID(idNumber);
    }
    public void setID(int idNumber)
    {
        id = idNumber;
    }
    public int getID()
    {
        return id;
    }
}
```



The extends keyword

```
public class Derived_Class_Name extends Base_Class_Name
{
    Declaration_of_Added_Instance_Variables
    Definitions_of_Added_And_Overridden_Methods
}
```

```
public class Student extends Person
{
    // stuff goes here
}
```

- A derived (child) class inherits the public instance variables and public methods of its base (parent) class

private vs. public

- private instance variables and private methods in the base class are NOT inherited by derived classes

- This would not work:

```
public Student(String stdName, int idNumber)
{
    name = stdName; // ERROR! name is private to Person
    setID(idNumber);
}
```

private vs. public

- private instance variables of the base class CAN be accessed by derived classes using the base class' public methods

- This works:

```
public Student(String stdName, int idNumber)
{
    setName(stdName); // OK! setName is a public method in Person
    setID(idNumber);
}
```

The super keyword

- A derived class does not inherit constructors from its base class
- Constructors in a derived class invoke constructors from the base class
- Use super within a derived class as the name of a constructor in the base class (superclass)
  - E.g.: super(); or super(intialName);
  - Person(); or Person(intialName) // ILLEGAL
  - First action taken by the constructor, without super, a constructor invokes the default constructor in the base class

this v.s. super

```
public Person()
{
	this(“No name yet”);
}
public Person(String initialName)
{
	name = initialName;
}
```

- When used in a constructor, this calls a constructor of the same class, but super invokes a constructor of the base class

Overriding Methods

- What if the class Person had a method called printInfo?

```
public class Person
{
    // a bunch of other stuff
    // ...
    public void printInfo()
    {
        System.out.println(name);
    }
}
```

Overriding Methods

- What if the class Student *also* had a method called printInfo?

```
public class Student extends Person
{
    // a bunch of other stuff
    // ...
    public void printInfo()
    {
        System.out.println("Name: " + getName());
        System.out.println("ID: " + getID());
    }
}
```

Overriding Methods

- If Student inherits the printInfo() method *and* defines its own printInfo() method, it would seem that Student has two methods with the same signature
  - We saw before that this is illegal, so what's the deal?

Overriding Methods

- Java handles this situation as follows:
  - If a derived class defines a method with the same name, number and types of parameters, and return type as a method in the base class, the derived class' method ***overrides*** the base class' method
  - The method definition in the derived class is the one that is used for objects of the derived class

Overriding Methods: Example

- Both Person and Student have a printInfo() method

```
Student std = new Student("John Smith", 37183);
std.printInfo(); // calls Student’s printInfo method,
                 // not Person’s
```

- Output would be:

```
Name: John Smith
ID: 37183
```

Overriding vs. Overloading

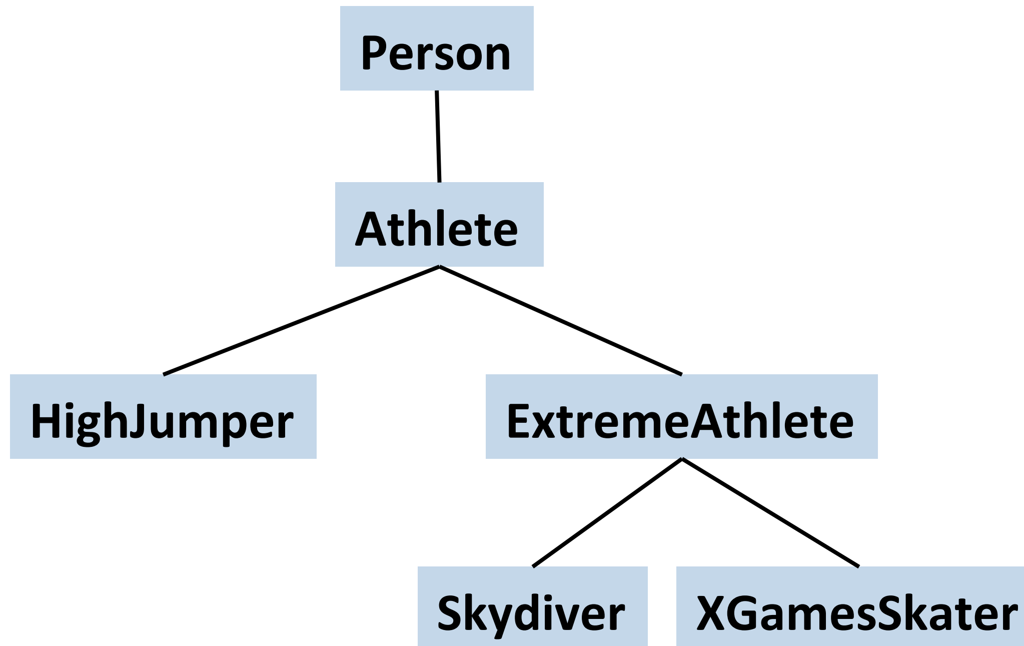
- If a derived class defines a method of the same name, same number and types of parameters, and same return type as a base class method, this is ***overriding***
- You can still have another method of the *same name* in the same class, as long as its number or types of parameters are different: ***overloading***

The final Modifier

- A final method cannot be overridden
  - E.g.: public final void specialMethod()
- A final class cannot be a base class
  - E.g.: public final class myFinalClass { … }
  - public class ThisIsWrong extends MyFinalClass { …} // forbidden

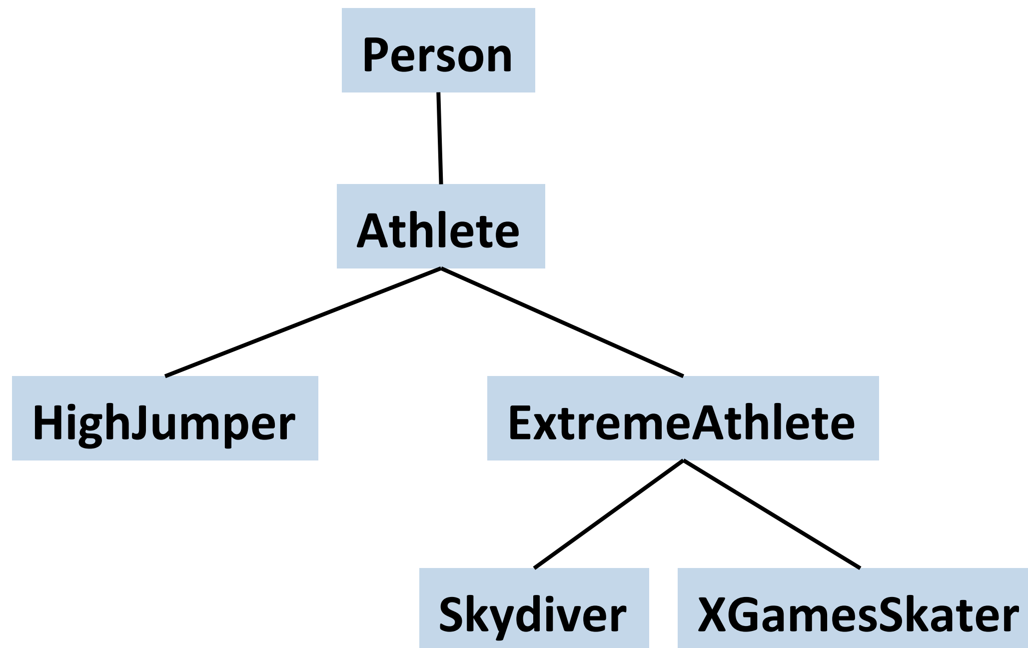
Type Compatibilities

- Given this inheritance hierarchy…



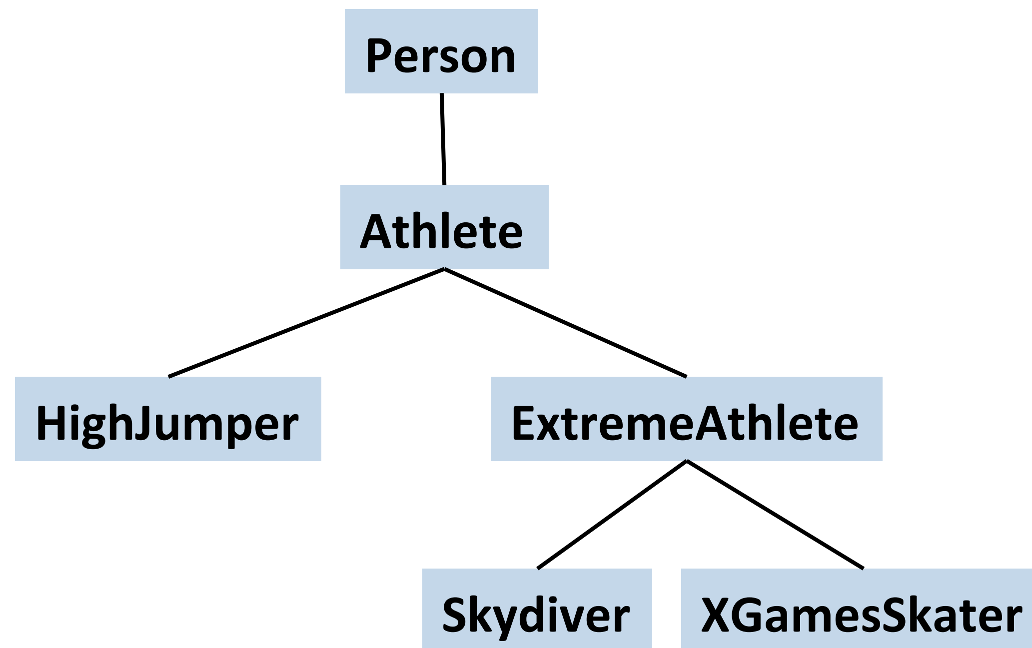
Is This Code Legal?

- `Person per = new Person();`
  - Yes!



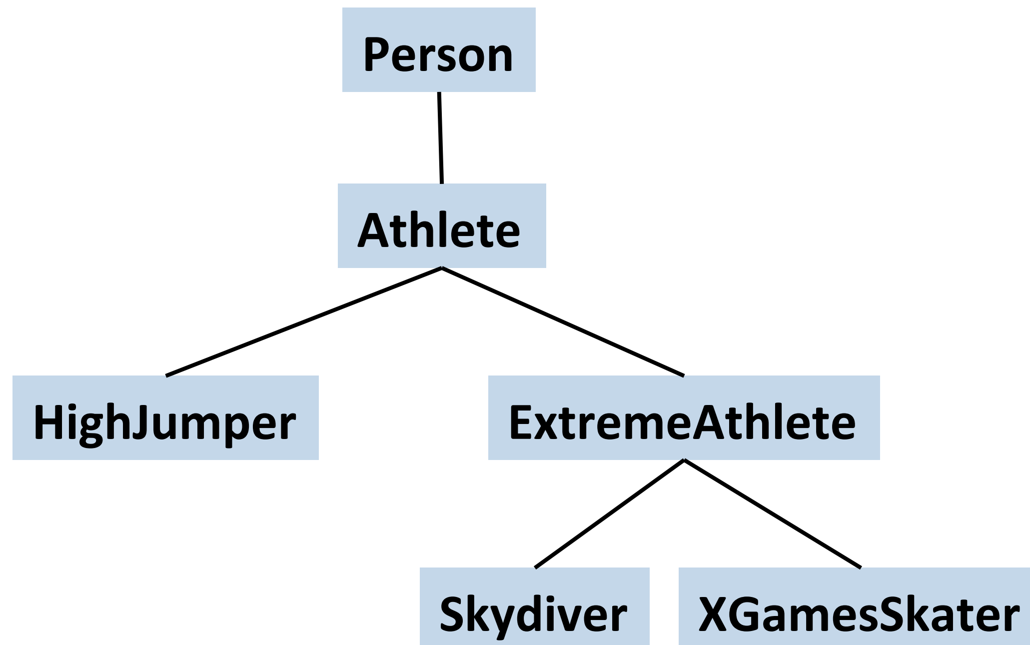
Is This Code Legal?

- `HighJumper hJumper = new HighJumper();`
  - Yes!



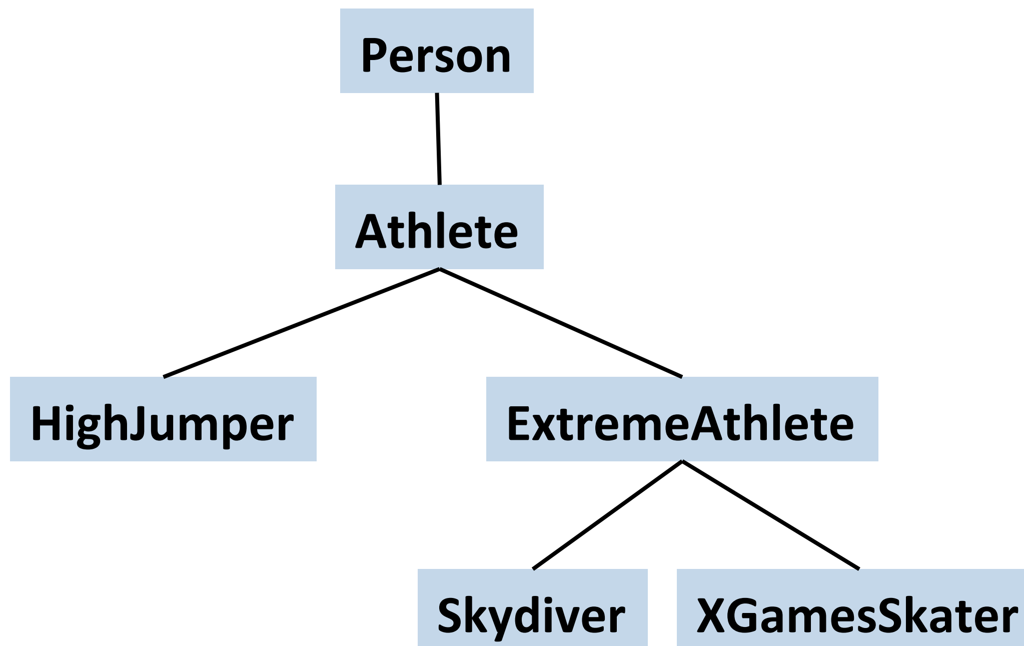
Is This Code Legal?

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



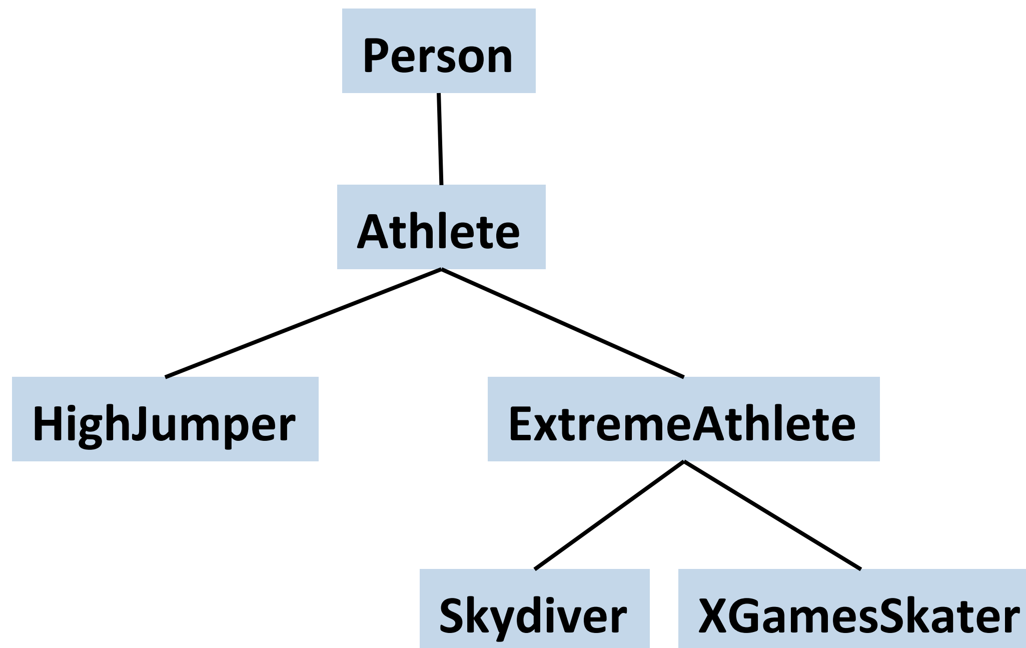
Is This Code Legal?

- `Skydiver sDiver = new Person();`
  - No! A Person ***is not necessarily*** a Skydiver, so this is illegal



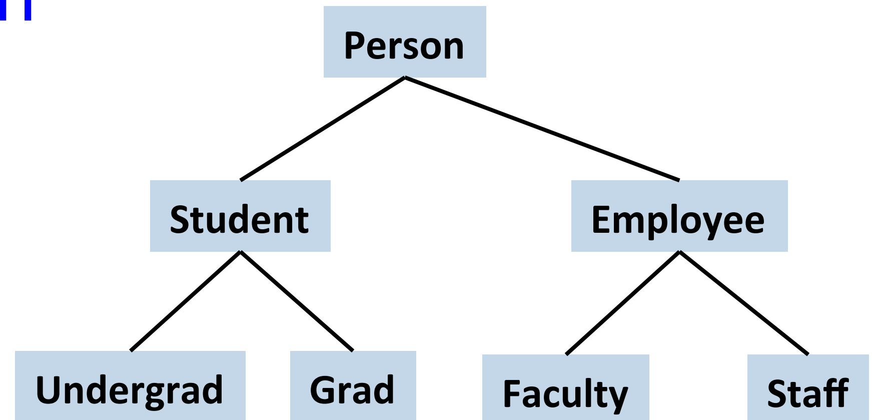
Is This Code Legal?

- ```
  Athlete ath = new Athlete();
  XGamesSkater xgs = ath;
  ```
  - No! An Athlete ***is not necessarily*** an XGamesSkater, so this is illegal


  ```

Summary

- An object of a derived class can serve as an object of the base class
- An object can have several types because of inheritance
  - E.g: every object of the class Undergraduate is also an object of type Student, as well as an object of type person



Next Class

- Inheritance and Polymorphism