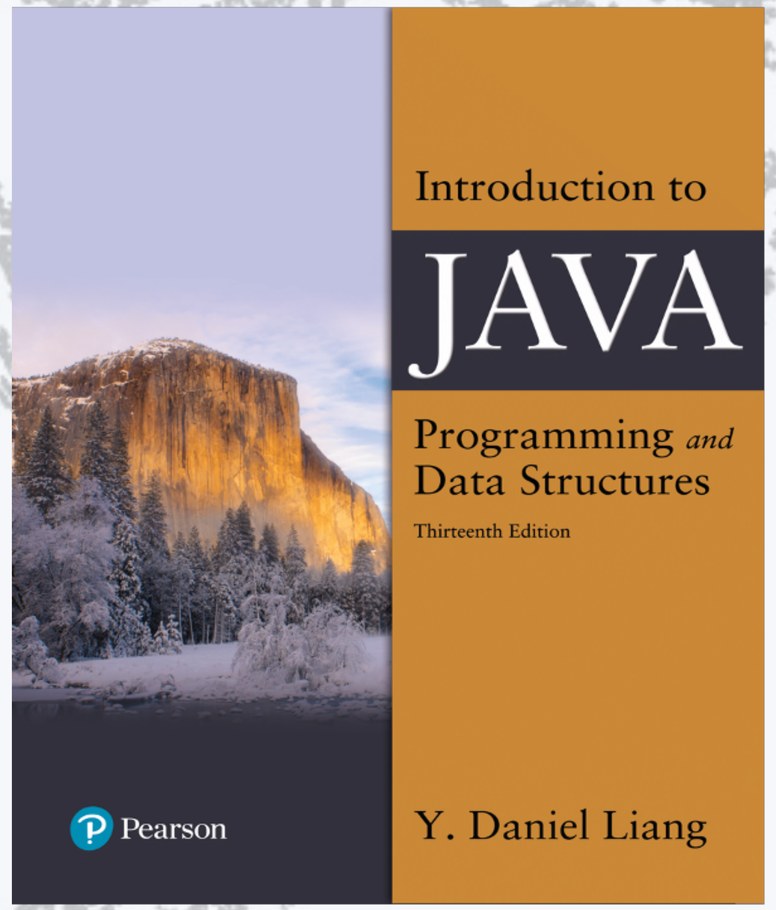


Inheritance and Polymorphism

Inheritance and Polymorphism

- Inheritance
- Polymorphism

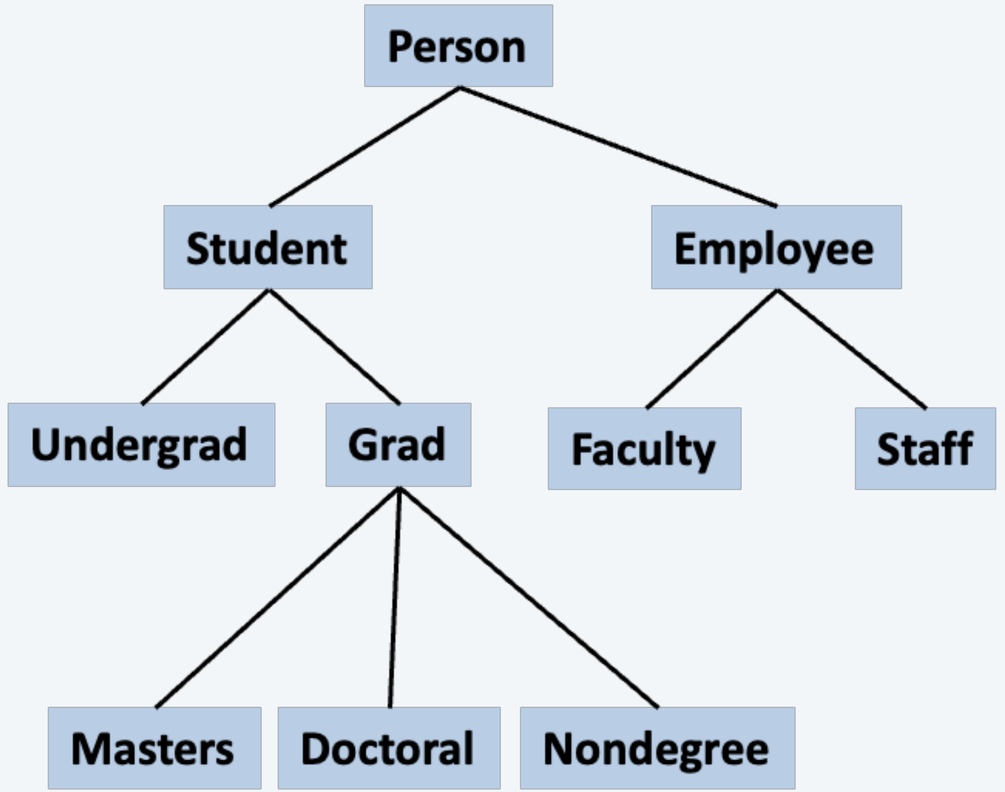
Motivations

Suppose you will define classes to model *ug students*, *pg students*, and *faculties*. These classes have many common features. What is the best way to design these classes so to avoid redundancy? The answer is to use **inheritance**.

Inheritance

Classes (or ADTs) can have relationships.

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

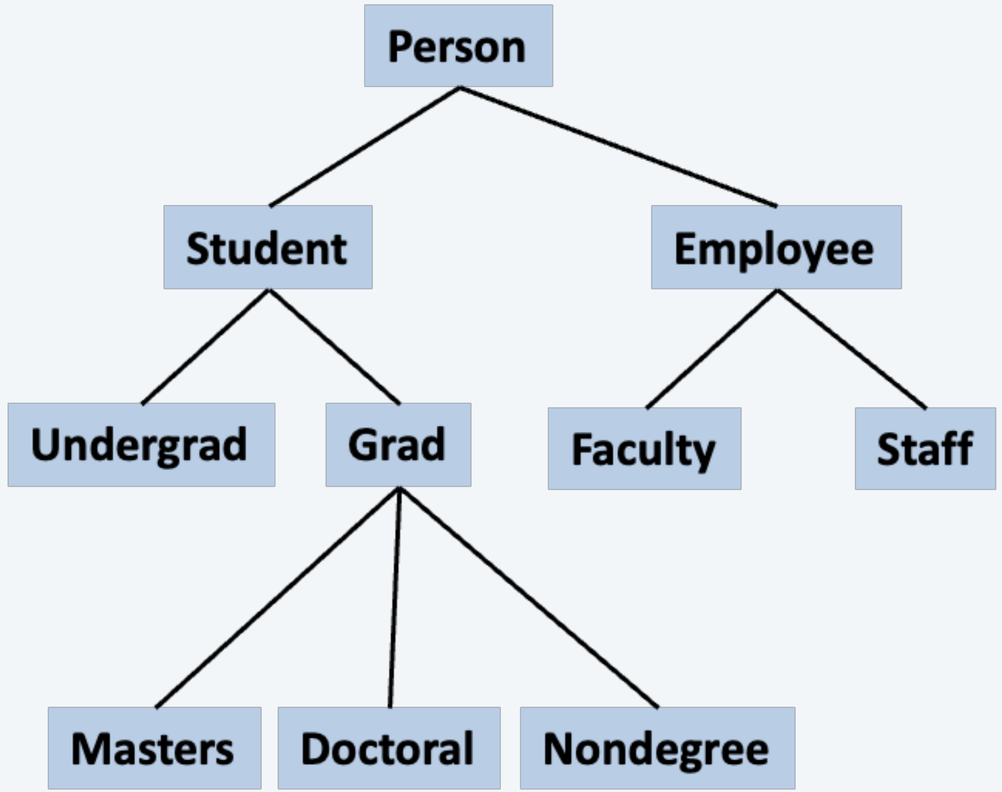


Inheritance

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

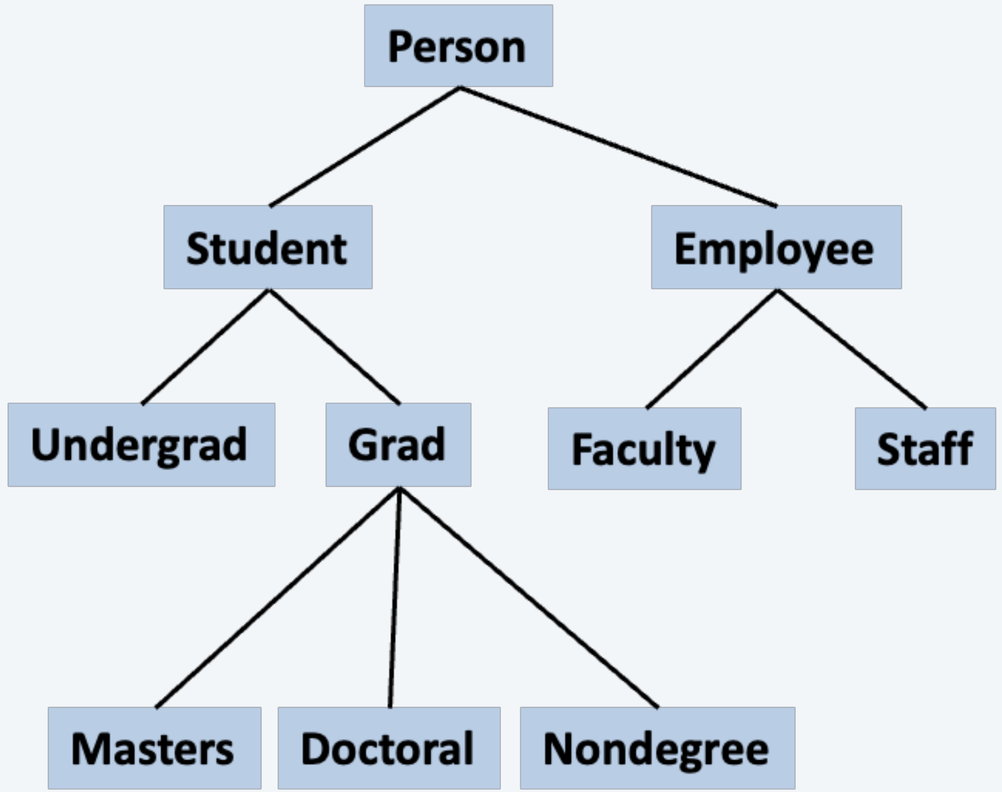


The *is-a* Relationship

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

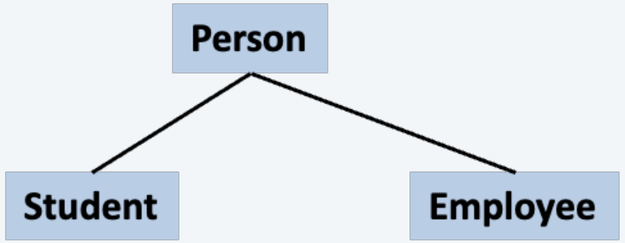
Is a Person a Student?
- Not necessarily!

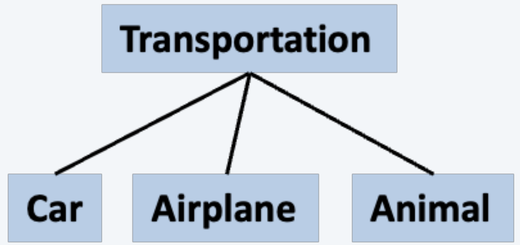


Base Class and Derived Class

- Our general class is called a ***base class***
  - Also called a **parent class** or a **superclass**
  - Examples: Person, Transportation

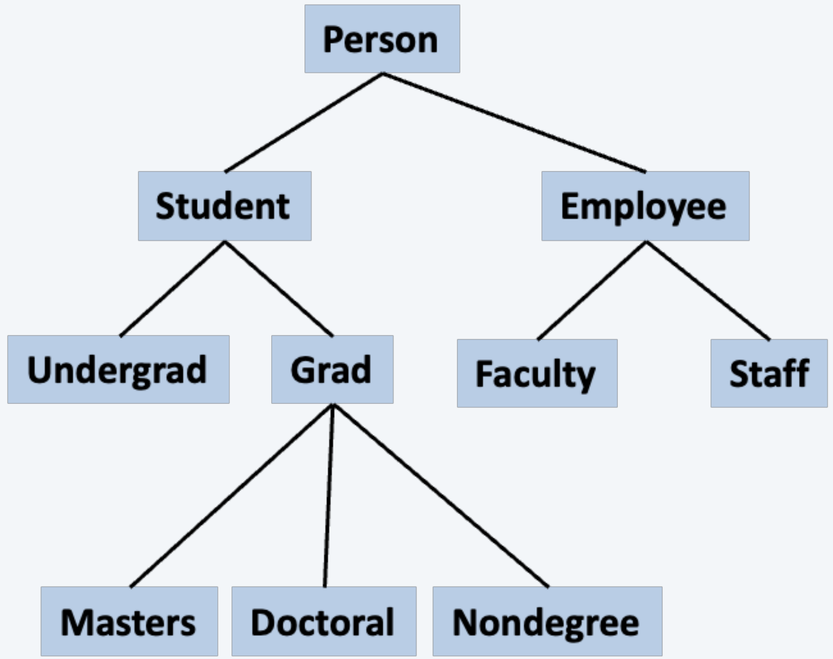
- A specialized class that inherits properties from a base class is called a ***derived class***
  - Also called a **child class** or a **subclass**
  - Examples: Student *is-a* Person, Employee…





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

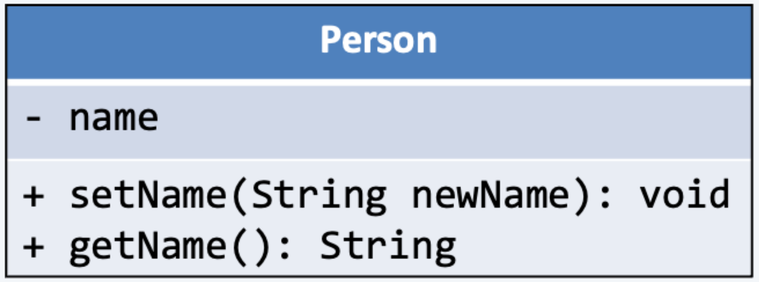


Why is Inheritance Useful?

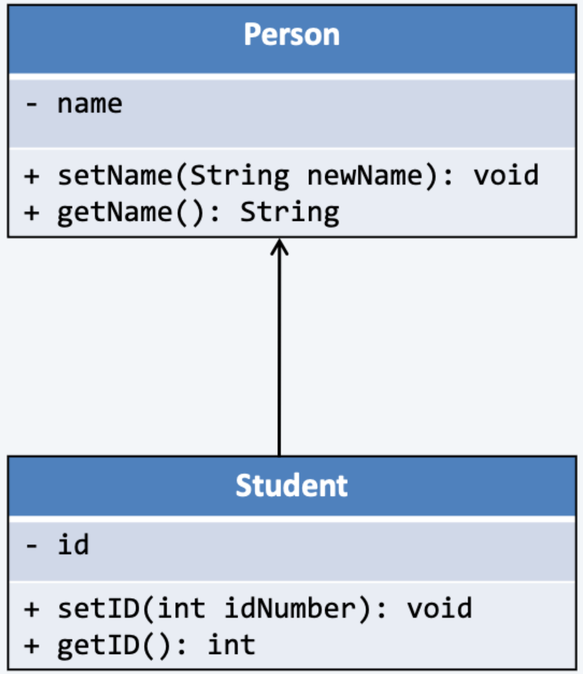
- Enables you to define shared properties and actions **once**
- Derived classes can perform the same actions as base classes without having to redefine the actions
  - If desired, the actions can be redefined – more on this late

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;
    }
}
```




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

```
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

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

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

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);
    }
}
```

What if the class Student 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?

Java handles this situation as follows:

- If a derived class defines a method with t**he 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

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);
    }
}
```

What if the class Student 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());
    }
}
```

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

Expected Output:

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

Overriding v.s. 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**

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

The final Modifier

A final method cannot be overridden

- E.g.: `public final void specialMethod()`

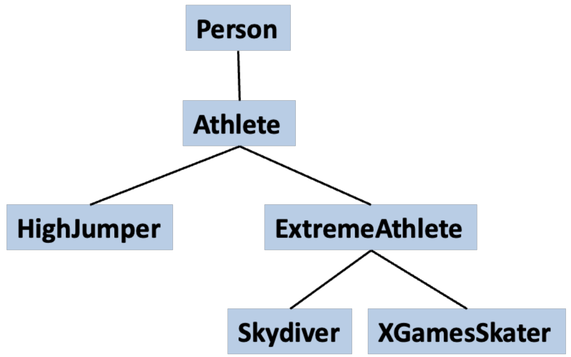
A final class cannot be a base class

- E.g.: `public final class myFinalClass { … }`
- `public class ThisIsWrong extends MyFinalClass { …} // forbidden`

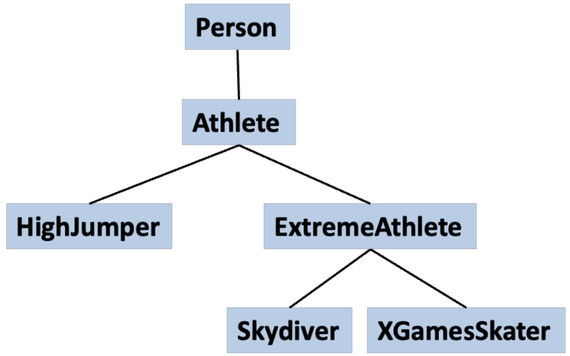
Pop up quiz

Is this code legal?

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



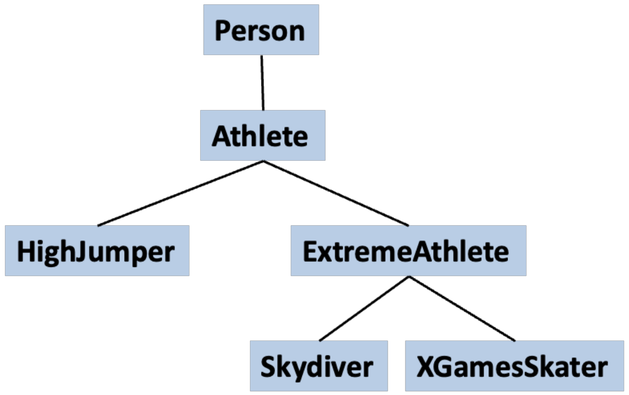
- `HighJumper hJumper = new HighJumper();`
  - Yes!



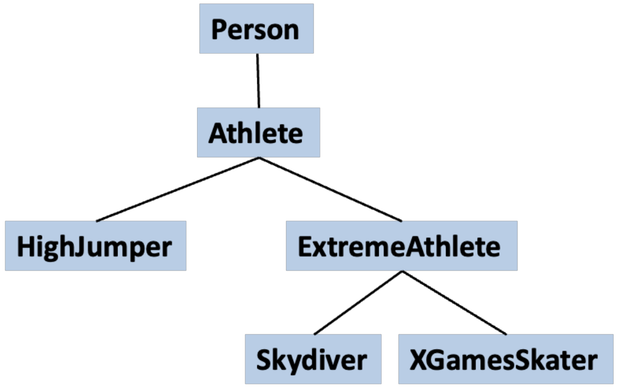
Pop up quiz

Is this code legal?

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



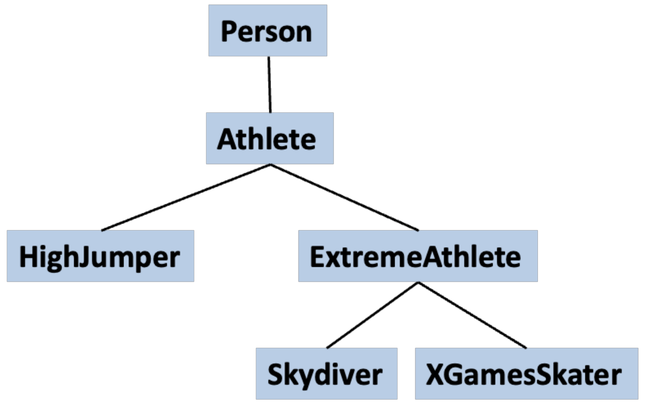
- `Skydiver sDiver = new Person();`
  - No! A Person *is not necessarily* a Skydiver, so this is illegal



Pop up quiz

Is this code legal?

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

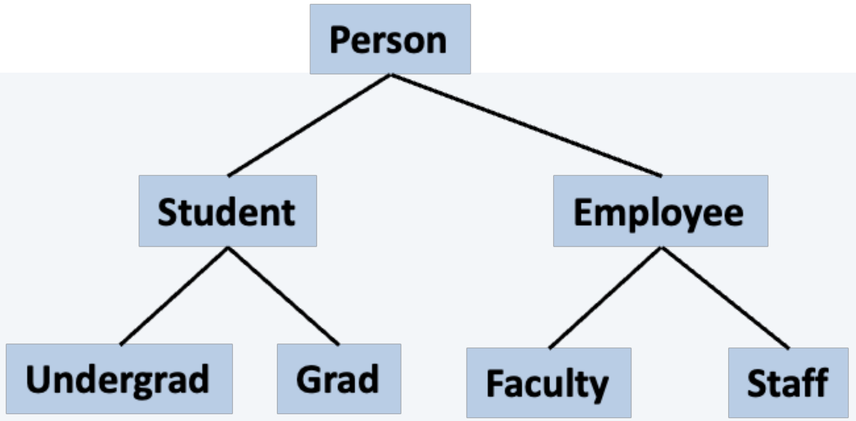

  ```

Summary of type compatibilities

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



Inheritance and Polymorphism

- Inheritance
- Polymorphism

Polymorphism

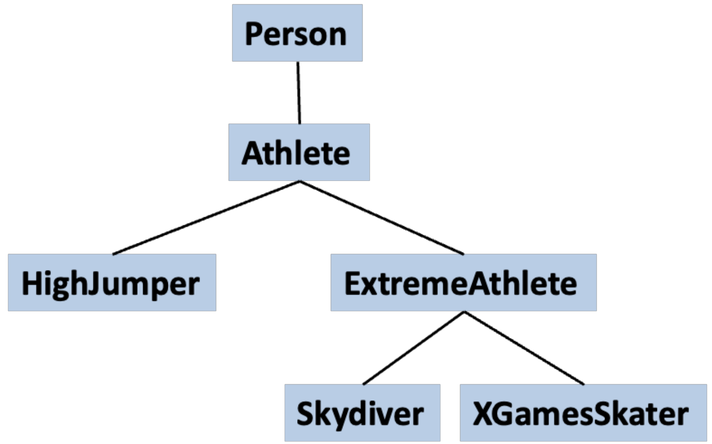
Inheritance allows you to define a base class and derive classes from the base class

Polymorphism means that a variable of a supertype can refer to a subtype object.

Calling a Derived Class's 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);
}
```



- 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***

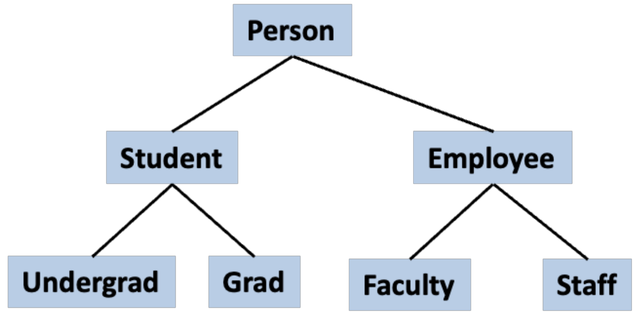
Dynamic binding

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.*

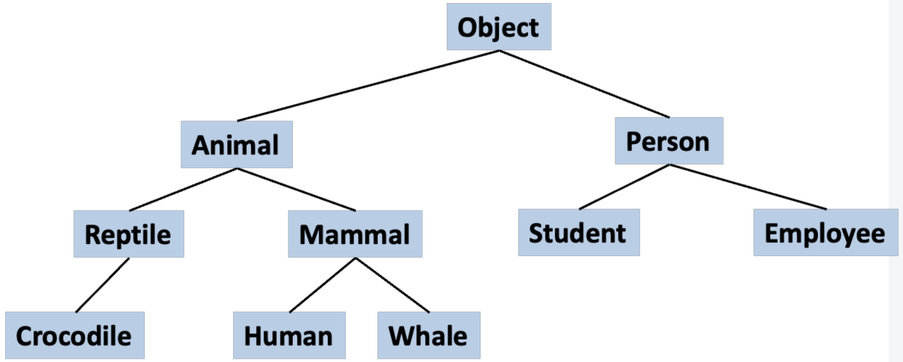
Dynamic Binding and Polymorphism

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



- 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

```
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
```

For 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

For derived class

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

Would this compile?

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

- 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

```
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 equals method

- next 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

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

```
object instanceof Class_Name
```