

Chapter 7

Inheritance

Class Inheritance:
Introduction
extends usage
protected visibility
abstract classes/methods

Hello!

Be sure to actually work through the examples in order; the "Your Turn" problems build upon each other extensively.

Today we will cover another key OO concept: inheritance. We're comfortable with all the structure and hierarchy of creating separate classes, using fields to get some aggregation going, but our class definitions are still all written separately. Inheritance is going to allow us to write classes by essentially borrowing all of another class's definition, and selectively adding more fields and methods to it, with some nifty relationships holding between the original and newly-defined classes.

Motivation

So far, the only way we could cause two classes to interact in any way was through aggregation: where the object of one class "HAS-A" object of another class as part of its own data. Let's quickly review an example of aggregation, because we need to be sure we contrast the idea of aggregation with the new concept of inheritance.

Aggregation Example: A Sphere HAS-A Coordinate:

```
public class Coordinate {
    public double x,y,z;

    public Coordinate (int a, int b, int c) {
        x=a;
        y=b;
        z=c;
    }
}

public class Sphere {
    private Coordinate location;
    private int radius;

    public Sphere (int r, int x, int y, int z){
        radius = r;
        location = new Coordinate(x,y,z);
    }
}
```

A particular sphere **has** its own location. Each time we create a `Sphere` object, we will have a `Coordinate` object as well. Although we see that an instance of one class (a `Sphere`) has a reference to an instance of another class (a `Coordinate`), neither class definition is a more specific version of the other. A `Sphere` isn't a `Coordinate`, even though a `Sphere` has-a `Coordinate`; a `Coordinate` isn't a `Sphere`. Similarly, `Sphere` isn't a `double`, even though it may have one to represent the radius.

Inheritance

The classes themselves each represent a type that is entirely separate from anything else: A `Sphere` object has-a radius, and it has-a `Coordinate` object, but nothing relates our `Sphere` class to a `Shape` class or a `Cube` class.

We want to start identifying different class definitions that should somehow be linked together, defined somehow via the same code.

When Would I Want Inheritance?

Consider the following three class definitions. We are creating a (very simple) program to use on campus, and want to store information about various people on campus. We have students, employees, and other non-specific persons.

```
public class Person {
    private String name;
    private int age;

    //methods: (constructors), getName, setName, getAge, setAge.
}
```

```
public class Student {
    private String name, major;
    private int age, masonID, yearsOnCampus;

    /* methods:
       (constructors),
       getName, setName, getMajor, setMajor,
       getAge, setAge, getMasonID, setMasonID,
       getYearsOnCampus, setYearsOnCampus.
    */
}
```

```

public class Employee {
    private String name, jobTitle;
    private int age, masonID, yearsOnCampus;

    /* methods:
       (constructors),
       getName, setName, getJobTitle, setJobTitle,
       getAge, setAge, getMasonID, setMasonID,
       getYearsOnCampus, setYearsOnCampus.
    */
}

```

We see some overlap in these definitions: all of these classes have a **name** and **age**. Some of the classes have **masonID** and **yearsOnCampus** variables. And a few other variables are unique to different classes (**major**, **jobTitle**). Now imagine that we added constructors, getters, and setters for each of these instance variables. We'd see more and more duplicated code, both in data and behaviors (methods).

We can create multiple objects from each of these classes:

```

Student s1 = new Student( "Bob", "art", 19, 71922, 2 );
Student s2 = new Student( "Helen", "education", 18, 71923, 1 );

Person[] people = { new Person("A",1)
                    , new Person("B",2)
                    , new Person("C",3)
                    };

Employee e1 = new Employee ("Catherine", "cashier", 35, 71985, 7);

```

But there's a problem: we can't actually use these definitions together very well. Suppose we wanted to represent the people in line at the cash register in the food court:

```

Person pers = new Person ("Sally", 25);
Student stud = new Student ("Buddy", "undeclared", 21, 71234,2);
Employee emp = new Employee ("Doug", "HR", 41, 72468, 10);

SomeType[] customers = {pers, stud, emp}; // BAD CODE
????????

```

What type should we put for `SomeType`? We can't choose any of `Person`, `Student`, `Employee`, because there's something in the array that doesn't match the type.

The problem is the lack of a relation between these different types: there is no way for us to treat a `Student` like a `Person`. Even though a `Student` has all the instance variables a `Person` has, and has all the methods a `Person` has, we can't treat a `Student` like a `Person`. Java has no idea that it is safe to treat a `Student` like a `Person`. How impersonal! Similarly, we can't treat an `Employee` like a `Person` either. Doug in HR is going to hear my complaints about this work environment.

We want to give Java the information so that these different classes are linked together: we want to be able to say a `Student` IS-A `Person`, and that an `Employee` IS-A `Person`.

So: we want to relate these different class definitions by actually defining one class in terms of another. We want to **reuse** the Person definition, and extend it to tell Java how a Student is a more-specific version of a Person, and how an Employee is a more-specific version of a Person.

```
public class Person {  
    private String name;  
    private int age;  
    //methods: (constructors), getName, setName, getAge, setAge.  
}
```

```
public class Student extends Person {  
    // we inherit name and age.  
    private String major;  
    private int masonID, yearsOnCampus;  
  
    /* methods that we write:  
       (constructors),  
       getMajor, setMajor,  
       getMasonID, setMasonID,  
       getYearsOnCampus, setYearsOnCampus.  
    */  
}
```

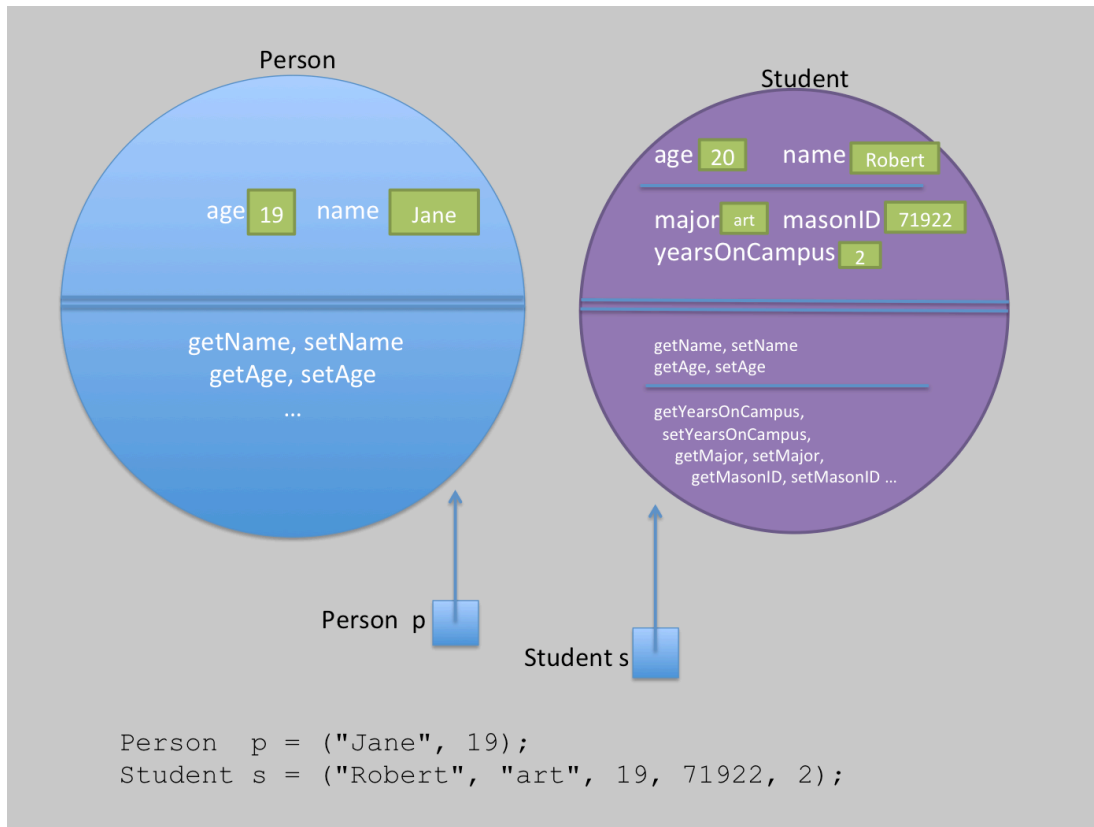
```
public class Employee extends Person {  
    // we inherit name and age.  
    private String jobTitle;  
    private int masonID, yearsOnCampus;  
  
    /* methods:  
       (constructors),  
       getJobTitle, setJobTitle,  
       getMasonID, setMasonID,  
       getYearsOnCampus, setYearsOnCampus.  
    */  
}
```

Let's stop and think a moment about our two uses of extends Person.

Student extends Person. By adding those two words (extends Person), we are telling Java that a Student is a Person. That means that all the data a Person has, a Student has that data, too. All the behaviors that a Person can exhibit, a Student can exhibit too. We don't explicitly list String name and int age in the Student class, but each Student object will have these instance variables, because a Student is a Person and a Person has a name and age. Similarly, all the methods that were defined in the Person class (e.g., getName, setAge) are also available to Student objects, because a Student is a Person.

When we draw the memory usage for objects on the whiteboard with a circle encompassing all the data and methods, we would now draw a Student object by drawing a Person object, and then adding in the major, masonID, and yearsOnCampus instance variables to the data portion (top half), and adding the extra

method definitions to the behaviors section (bottom half). Again, note that all we do with subclasses is add definitions, never take away. This crucial only-adding-things property is why we're able to say that every single `Student` object can always be used as a `Person` object. It's as if we temporarily ignore the extra variables and methods that `Students` have, and just rely on the inherited portions.



In the diagram, we see that the `Student` object has all the `Person` fields (`age`, `name`), followed by the extra fields specific to `Students` (`major`, `masonID`, `yearsOnCampus`). Similarly, it has all the `Person` methods, followed by all the specific `Student` methods (getters/setters for `major`, `masonID`, and `yearsOnCampus`). If we ignored parts of the `Student` object, it would look just like a `Person` object. This is the guarantee that Java is given when we extend a class to create a more specific version of it.

Some Terminology

We have a lot of descriptive ways to describe the relationship between `Student` and `Person`.

- We can say that the `Student` class is a subclass of the `Person` class, or that `Person` is a superclass of `Student`.
- We can also call `Student` a child-class of `Person`, and `Person` the parent-class of `Student`.
- We can also say that `Student` is a subtype of `Person`, and `Person` is a supertype of `Student`.

Using the is-a Relationship.

So how can we actually use this "Student is-a Person" relationship? Consider the following legal code (once we've actually written the constructors):

```
Person p = new Person ("A",1);
Student s = new Student("B", "CS", 20, 71235, 3);
```

```
System.out.println("p name: " + p.getName());
System.out.println("s name: " + s.getName());
```

```
p = s;
System.out.println("p name: " + p.getName());
```

The `p` variable can only hold a reference to a `Person` object. But a `Student` is-a `Person`, so `p` can also hold a reference to a `Student`, because that means it also happens to be storing a reference to something that is-a `Person`.

Now, back to our register example. We now know that a `Person` is-a `Person`, a `Student` is-a `Person`, and an `Employee` is-a `Person`. So it's safe to use an array of `Person` references to store our various values:

```
Person[] customers = {pers, stud, emp};
```

Your Turn!

Create files for all three of the above classes. For now, don't write constructors yet; rely on the annoying default constructors. That means we have to create our objects this way:

```
Person p = new Person();
p.setName("Bob");
p.setAge("20");
```

```
Student s = new Student();
s.setName("Jane");
s.setAge(19);
s.setMajor("art");
...
```

- Caution: if you accidentally add `name` and `age` fields to `Student` or `Employee`, Java will allow it (it is called [shadowing](#), and is generally a bad idea to use this 'feature'). Make sure that you don't re-define anything that is supposed to be inherited.
- Add another class that extends `Person`. Choose some other type of person that would be on campus: `Policeman`, `Athlete`, `Professor`, or whatever. Even if it seems to overlap with either `Student` or `Employee`, it's okay, as long as there's extra data you want to keep track of for this new type of person.
- Create objects of each of your class types. (Put this code in `TestInheritance.java`). Explore using the getters/setters, and see that you do indeed get to use methods declared in the `Person` class when using a `Student` object, `Employee` object, and so on.
- Create a `toString` method for **just** the `Person` class, and not the others. Retest the "Using the Is-A Relationship" code, but just print out `p` and `s` instead of `p.getName()` and so on. Notice that even methods like this can be inherited. We'll see how we can do better to represent `Student` objects than being stuck with the `Person` version of the `toString` implementation.

Visibility

We finally have a chance to experience the last visibility: **protected**. Try writing a `toString` method for the `Student` class. Print all the relevant info. For example, try to get your `toString` method in the `Student` class to output:

```
Student{name="foo", major="art", age=19, masonID=1234, years=2}
```

What goes wrong? We get a complaint from the Java compiler that `name` and `age` are `private`. (Assuming you used the visibilities shown above! If the knee-jerk reaction to make everything public kicked in, just go back to `Person`, make those instance variables private, and then see the child fail to access it). Even though we're inheriting from the `Person` class, *private* still means "only accessible in code actually written in this class definition". Rather than make `name` and `age` *public* (which violates encapsulation principles), we are offered a compromise: the **protected** visibility. This will cause `name` and `age` to still behave as if it were `private` in places such as your testing code (in some other class like `TestInheritance`), but they will now behave as if they were `public` when accessed from a child class like `Student`.

Your Turn!

- Change the visibility of `name` and `age` to `protected`.
 - First, verify that you still can't access them from `TestInheritance`. This is because it's not a child class of `Person`.
- Go back and write your `toString` method in the `Student` class. Now that they are protected, it's as if they were declared with `public` visibility as far as the child classes are concerned.
 - Revisit the "Using the Is-A Relationship" example from above (and still just print `p` and `s` rather than `p.getName()` and so on). Now it's even more apparent that we have a `Student` object being used where we expect a `Person`.

Overriding Methods

In the last example, we witnessed something that is actually kind of amazing: the `Student` class got to **override** the definition of the `toString` method. `Person` already provided a `toString` method, which used to be what our `Student` class used. But we decided we could do better, and got rid of that definition in favor of a more specific one.

We can override methods inherited from the parent class, with a few requirements:

- We have the *exact same method signature*, and then we get to supply a different body (implementation) for the method.
- The parent class has to give us permission to override a method by not making the method **final**. Yes, methods can be `final`, too – it means that the method body cannot be modified. So if we don't want re-implementations of a method in child classes, we just say something like

```
public final int importantMethod(The args)
```

and now overriding is not allowed.

Whenever our code calls this overridden method on objects of our child class, it will always use this 'better' version that we provided, thus overriding the original definition.

Your Turn!

- Add the `wakeUpTime` method to the `Person` class, and override it in the `Student` class. (Just return a `String` for whether this person wakes up early or late; I'm assuming that students like to sleep in, and employees don't get to).
- Test it out (call it on a `Person`, and on a `Student`).
- Now make the `Person` version of the method `final`, and look for the compiler error.

We can actually tell Java that we intend to override a method: just put `@Override` right before a method that is supposed to be an overriding definition. If anything goes wrong, such as not being allowed to override the method, or it turns out that your method doesn't actually override any method, the compiler can now alert you.

→ when would we not be overriding a method when we thought we were? Consider a method with signature `@Override public String toString()`. It's trying to override `toString`, but neglected to capitalize the `S`.

Your Turn!

- add a method that isn't inherited to your `Student` class. Add the `@Override` tag and see that the Java compiler catches the issue.

Inheritance and Constructors

Let's finally discuss how to make constructors for child classes. As a first step, add a nice constructor to the `Person` class:

```
public Person (String name, int age) {  
    this.name = name;  
    this.age = age;  
}
```

You will also need to change your `Person` instantiations to look like `new Person ("Bob", 20)`. Try to recompile. Specifically, try to recompile just the `Student` class:

```
javac Student.java
```

What happens? Java can create `Person` objects using the constructor you added, but now there's no default constructor, and so we can't even compile the `Student` class anymore, which was actually relying on it. Our `Person` class declared that it was no longer acceptable to use the default constructor, and it exposed the fact that our `Student` constructor actually used the `Person` constructor to complete the task of constructing a `Student` object. Let's create a constructor for the `Student` class.

First (Bad) Attempt. Let's try to just write a `Student` constructor that doesn't rely on the `Person` constructor at all.

```
//BAD attempt at a child class constructor
public Student (String name, String major, int age, int masonID, int years){
    this.name = name;
    this.major = major;
    this.age = age;
    this.masonID = masonID;
    this.yearsOnCampus = years;
}
```

This fails; somehow, the Java compiler still wants to find the constructor `Person()`. It turns out we have to learn another keyword: **super**. `super` is similar in nature to `this`. The keyword `this` refers to the current object, giving us access to values and methods of the object. `super` refers to the parent class, giving us access to the members of the parent class which we otherwise maybe couldn't see in the child class. If we want to access anything in the parent class, such as a specific constructor method, we have to use `super` to access it.

```
// GOOD attempt at a child constructor
public Student (String name, String major, int age, int masonID, int years){
    //call the parent constructor
    super(name,age);
    //finish instantiating things declared in this class.
    this.major = major;
    this.masonID = masonID;
    this.yearsOnCampus = yearsOnCampus;
}
```

The call `super (name, age)` is actually calling the constructor of the `Person` class that accepts a `String` and an `int`. The call to `super` needs to be first in our constructor body so that Java knows what's going on.

Required super calls

Java **requires** us to use the parent class's constructor when writing our own constructor. This ensures that our claim that "Every `Student` is a `Person`" won't be violated by some naïve implementation of a constructor. It turns out that an implicit **super()** call is added to constructor method of child classes unless we add our own `super`-call. That's why our first attempt at a `Student` constructor failed: there was no `public Person()` constructor definition any more.

By creating a new constructor for `Person`, and then explicitly calling that new constructor with a call to `super (some, args)`, we are fulfilling the "always use your parent class's constructor" requirement while also completing the instantiation efforts that are local to our own class's instance variables.

Your Turn!

- Create constructors for all classes. Be sure to use the non-default constructor from the `Person` class by using the correct super call to the `Person` constructor from your child classes' constructors.
- When creating constructors for `Student` and `Employee`, remember that you have five instance variables to instantiate (two came from the `Person` class, three are defined here in the class).
- Create all the getters and setters for the instance variables introduced in each class. (Remember, the ones defined in the `Person` class are inherited in the child classes: you won't write `getName` in the `Student` class, but `Students` will inherit the `getName` getter that you define in the `Person` class). Will anything need to be made `protected` instead of `private`?

Abstract Classes

One last topic for inheritance is the notion of an **abstract class**. Abstract classes participate in the class hierarchy of parent and child classes, but by making a class abstract, we are stating that no objects of this specific class may ever be created. (Objects of child classes would be permitted; otherwise, there would be no point!).

Let's consider an example of an abstract class. If we look to our original three classes, we saw a slight chance for more overlap: the `Student` and `Employee` classes shared instance variables `masonID` and `yearsOnCampus`. Let's create a class to help us indicate that relationship by leaving the `Person` class alone, but introducing a new class between the `Person` class and the two child classes `Student` and `Employee`:

```
public class MasonPerson extends Person {
    protected int masonID;
    protected int yearsOnCampus;

    public MasonPerson(String name,int age,int masonID,int years) {
        super(name, age);
        this.masonID = masonID;
        this.yearsOnCampus = years;
    }
}

public class Student extends MasonPerson {
    protected String major;

    public Student(String name, String major, int age, int masonID, int years){
        //calls MasonPerson constructor
        super(name,age,masonID,years);
        this.major = major;
    }
    //no more variables, only methods below...
}

public class Employee extends MasonPerson {
    protected String jobTitle;
    public Employee (String name, String job, int age, int masonID, int years){
        //calls MasonPerson constructor
        super(name, age, masonID, years);
        this.jobTitle = job;
    }
    //no more variables, only methods below...
}
```

One nice thing is the transitivity of inheritance: because a `Student` is-a `MasonPerson` and a `MasonPerson` is-a `Person`, we can transitively claim that a `Student` is-a `Person`. All the data/behavior that `MasonPerson` inherits from `Person` will be passed on to `Student`.

On to abstract classes. Suppose, for whatever reason, that creating a `MasonPerson` doesn't make sense: we only want there to be objects of specific kinds of people: `Student` objects and `Employee` objects are okay, but the vague idea of a `MasonPerson` only helps us organize our thoughts; any `MasonPerson` must actually be some more specific type. We then choose to make the `MasonPerson` class abstract, indicating that it is illegal to instantiate the class:

```
public abstract class MasonPerson extends Person { ... }
```

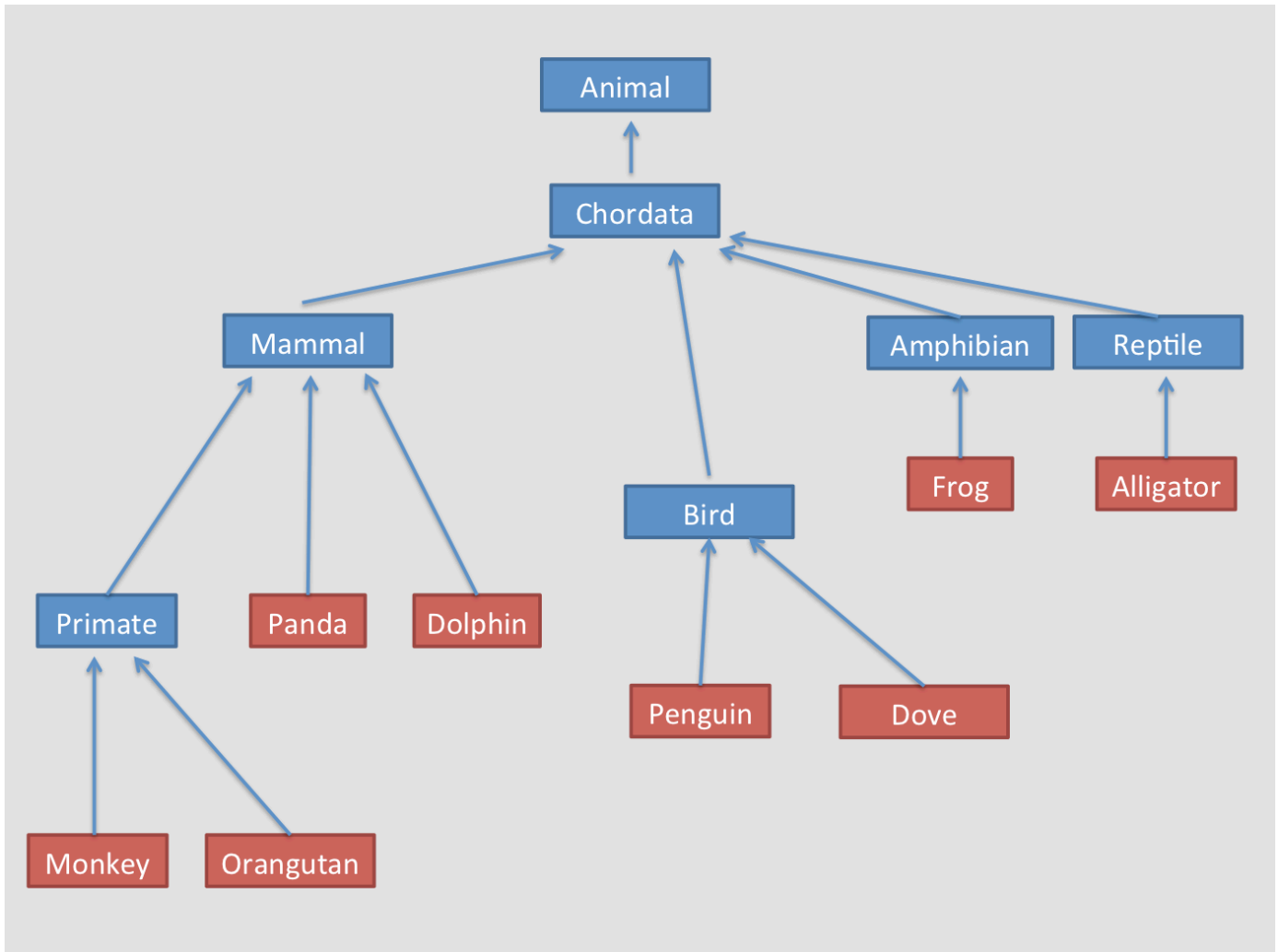
Your Turn!

- Add the `MasonPerson` class, which extends the `Person` class.
 - Add instance variables `masonID` and `yearsOnCampus`.
 - Give it a constructor, using the `Person` constructor.
- Change `Student` and `Employee` to be child classes of `MasonPerson`. Update their constructors to use the `MasonPerson` constructor.
- In your `TestInheritance` class, create a `MasonPerson` object and test it a bit.
- Make `MasonPerson` an abstract class; re-compile `TestInheritance` and see the error message.
→ you've successfully created an abstract class that contributes to the class hierarchy, yet is safely not instantiable (we can't create objects of it).

(keep going! Big chart + description all on the next page →)

Another Abstract Classes Example

An alternate example could be a program representing a zoo, where we use the Kingdom-Phylum-Class-Order-Family-Species hierarchy to organize our classes of animals. We want to have classes for Monkey, Orangutan, Panda, Dolphin, Penguin, Dove, Frog, Alligator and more animals. It makes sense to add classes to generate the following hierarchy:



It makes sense to have `Monkey`, `Panda`, and `Dolphin` objects, but it doesn't make sense to have a `Mammal` object or a `Chordata` object. Those classes just helped us organize things, and maybe provide definitions like `numVertebra`, `numChromosomes`, `eggSize`, and so on. If we make all the blue classes abstract, they still participate in the hierarchy of types (and can contribute fields and methods for inheriting), but are guaranteed to not be instantiated themselves.

More Abstract Things

We can use the `abstract` keyword for methods, too. What is an abstract method? It is a method signature with no body:

```
public abstract int reportNumberOfLegs();  
public abstract void moveAround();
```

Any time we want to indicate that every child class of our abstract class must have its own version of a method, yet there's not good starting implementation right now in the abstract class, we can create an

abstract method to require that all child classes provide implementations (by overwriting them). If they don't, they become abstract as well, guaranteeing that this method will be implemented for any actual instance of a class that is a child of the abstract class that introduced this abstract method.

- Any class containing an abstract method must be abstract as well.
- Any class that inherits an abstract method can implement it by overwriting that definition.
- Any class that doesn't implement an inherited abstract method therefore still contains an abstract method, and is thus also abstract.

Your Turn!

- Now that `MasonPerson` is an abstract class, add an abstract method to it, such as:

```
public abstract String favoriteFoodSite();
```

- Try instantiating a `Student` and see that it's now becoming abstract (well, we see compilation errors complaining to that effect).
- Override the abstract method that was inherited, and now we again are able to instantiate our `Student` class.