1 Review

Recall that inheritance is the mechansim that allows us to create class hierarchies in Java. It enables us to incrementally extend the functionality of a class without having to reimplement the existing functions.

In inheritance, we have a *base class* (aka superclass) from which other classes are derived. A *derived class* (aka subclass) is such that:

- it inherits all the properties of a base class (i.e. all public methods and fields are inherited, and have identical implementations)
- it can add data fields and additional methods (extending the class), and
- it can change the meaning of the inherited methods (overriding methods)

For example, our `Male` and `Female` class extends the `Person` class. The methods `toString()`, `getAge()`, and `rateDatingPotential()` are inherited, along with all the properties `name`, `height`, `mother`, etc. The `Male` class extends the `Person` class by adding properties `hrsWatchingSports` and `isBoxers` and method `rateInsensitivity()`. The `Female` class extends the `Person` class by adding properties `pairsOfShoes` and `bustSize`. And lastly, both the `Female` and `Male` classes override the `Person` method `rateDatingPotential()` with their own implementations.

2 Polymorphism and Casting

A derived class is type compatible with its base class. This means that a reference variable of the base class type can reference an object of the derived class. The converse does not hold!

```
Person p = new Female( "Barbie" );  // OK
Female f = new Person( "Pat" );     // Not OK
```

Why is this? Well suppose we now run the following code:

```
f.getName();                         // OK since f holds a Person
f.getPairsOfShoes();                 // Not OK: Person doesn't have getPairsOfShoes()
```

Since the variable `f` only contains an instance of the `Person` class, we will not be able to call all the methods that the `Female` class implements. On the other hand, when we assign an instance of the `Female` class to a `Person` variable, we are OK because anything we could call on a `Person` instance also exists for a `Female` instance (due to inheritance, yo!):

```
p.getName();                         // OK
p.getAge();                          // Also OK
p.getPairsOfShoes();                 // OK!  This is polymorphism!
```

3 Inheritance Hierarchies

4 Use of super

Suppose the class `C` is a class that inherits from class `A`. If used in class `C`, the keyword `super` refers to the superclass of `C`, i.e. `A`. So in an instance of class `Female`, `super` would refer to the superclass `Person`.

1. It's fairly standard in constructing a derived (sub) object to first construct the inherited part (the superclass' part) of the object and *then* construct the rest. In other words, in the derived class' constructor you first call your parent's constructor:

```
public Female( String n, int shoes ) {
    super( n );                 // Calls Person's constructor
    pairsOfShoes = shoes;       // Does other Female-specific initializations
}
```

   The `super` method can only be used in the first line of the constructor. If it is not explicitly called then an automatic call to the `super` with *no paramters* is generated.

2. You can also use `super` to call any of the base class' methods. I.e. suppose

```
public class Female extends Person {
    // Fields & Other Methods...

    // This will replace the already implemented rateDatingPotential
    // supplied by the Person class.
    public int rateDatingPotential() {
        int p = super.rateDatingPotential();  // First rate as a Person
        p     = p - pairsOfShoes/4;           // A lot of shoes is bad
        p     = p + bustSize/8;               // Rumour: the more bust the better
        return p;
    }
}
```

   And so `super` gives us a way to override the overriding method (by calling the base class' overridden method directly!)

5 Use of this

Suppose we have an instance `p` of class `Person`. Recall that class `Person` contains a method `getAge()`. Within the body of `getAge()` the keyword `this` refers to the instance `p`. There are many ways to use `this`:

1. You can precede a reference to any field or method in a class by `this` without changing its meaning: `this.toString()` ≡ `toString()`.

2. You can use `this` as an argument, to pass the name `p` as an argument: `bigger( this, e )`

3. You can use `this` to refer to another constructor:

```
public Female( String n, int shoes ) {
    this( n );              // Calls the existing constructor for just a name
    pairsOfShoes = n;       // Does other initialization with the other parameter
}
```

6 Abstract Classes

Suppose we have some method that we we want implemented in all our derived classes (our subclasses). So far, if this were the case then we would simply add this method to the base/super class and inherit the method down to the subclasses. But what if the method is such that you can't have an implementation in the super class because the method has no meaning there? I.e. what if we just *can't* implement this method in the base class because we don't have enough information at that point? Can we still guarantee that all the derived classes will have this method?

6.1 Boring but Effective Example

What the hell am I talking about? Are you as confused as I am? Well let me try to make things clearer. Suppose I have some base class `Shape` that represents a geometric shape. Furthermore, suppose we have subclasses `Circle` and `Rectangle`, both derived from `Shape`. A class `Square` could then be derived from `Rectangle`:

The `Shape` class may have fields that are commmon to all classes, such as `position` or `colour`, and it may have common methods such as `moveTo( int x, int y )` or `rotate( int degrees )`. You could imagine that these methods could be written in the superclass `Shape` and be inherited by the subclasses `Circle, Rectangle, Square`, etc. because their implementation doesn't depend on what kind of shape they are exactly. But how would I write an `area()` method? I can't! I can write one for `Circle` and `Rectangle` and `Square` but I can't write one for a generic `Shape` because it would have no meaning. So my solution is to declare this method to be abstract.

An *abstract method* is a method signature that declares the functionality that all derived objects must eventually implement. In other words, an abstract method is a method signature, but does not contain a body. It tells the derived type what return type it must return and what arguments it will have to take in when the derived type finally implements this method.

```
abstract class Shape {
    int    x, y;         // (x,y) coordinates of the shape -- gets inherited
    String colour;       // colour of shape -- gets inherited

    // setCoordinates is an ordinary method that gets inherited
    void         setCoordinates( int x, int y )  { this.x = x; this.y = y; }

    // area is an abstract method -- so derived class must eventually implement it
    abstract int area();
}
```

If a class contains an abstract method then the class itself is considered to be abstract. When a derived class doesn't implement all the abstract methods then the methods stays abstract in the derived class, and the derived class itself must remain abstract. In our shape example, the method `area()` would be declared `abstract` in the `Shape` class, and so the class `Shape` would need to be abstract as well. By putting in

this declaration we are saying that every type of `Shape`, whether it be `Circle`, `Rectangle`, or some other inherited `Shape`, must either implement the `area()` method or be considered abstract itself.

You cannot instantiate an abstract class because, by definition, not all of the methods exist. Suppose class `Circle` implements `Shape`'s abstract `area()`:

```
Shape circle, shape;
circle = new Circle( 3.0 );       // Legal because Circle is not abstract
shape  = new Shape( "circle" );   // Illegal because Shape is abstract
```

But even if you can't instantiate an abstract class, it can still have constructor for the derived classes to call for initialization purposes.

Question: Can you think of a reason to make the `Person` class abstract?

7 Multiple Inheritance

So far all of our inheritance examples have only extended from one class, (ex. class `Female` and `Male` both extend from `Person` only). Recall that we briefly discussed creating a class `Transvestite`, which we wanted to extend both classes `Male` and `Female`. We ran into a problem with the method `rateDatingPotential()` which is implemented differently in the `Male` and `Female` classes. If we tried to call this method from our new `Transvestite` class, Java wouldn't know whose method to call: `Male`'s or `Female`'s? And so Java doesn't allow multiple extends due to these naming and implementation conflicts. But we would still like to be able to inherit from multiple classes! The solution to this Earth-shattering problem is what we call an interface.

An *interface* is an extreme abstract class — it's a class where NO methods are implemented, only declared. The methods are called abstract because their bodies are not present; the bodies are replaced by semicolons. The methods are automatically public. An interface, like an abstract class, *cannot be instantiated* due to its incomplete specification.

```
public interface PersonInt {
    public String getName();
    public int    getAge();
    public int    rateDatingPotential();
}

public interface FemaleInt extends PersonInt {
    public int    rateCattiness();
}

public interface MaleInt extends PersonInt {
    public int     rateInsensitivity();
}
```

Note that there are no method modifiers (public, protected, private, static, ...) and there is no method body.

7.1 How do you use one of these things?

To use an interface we bring in the keyword `implements`. Using `implements` in the class definition means that the class MUST define all of the methods in the interface — it can't leave some blank like implementors of abstract classes. If you don't define all of the methods then you WILL GET a compilation error.

```
public class Transvestite implements FemaleInt, MaleInt {
    // Transvestite is a full-blown class, so I'll need some fields
    String name;
    int    height;
    int    pairsOfShoes;
    int    hrsWatchingSports;

    // It will need a constructor
    public Transvestite( String n )     { name = n; }

    // It will have to implement all the methods stated in
    // FemaleInt, and MaleInt --- no more implementation conflicts!
    String getName()                    { return name; }
    int    getAge()                     { ... }
    int    rateDatingPotential()        { ... }
}
```

7.2 Why is this Useful?

When we first talked about inheritance, we created a `Person` class that contained person properties and related methods. We wanted all `Person` instances to have the functionality we defined, and we were able to guarantee the users of a `Person` class that we had this functionality. But we ran into a problem when we implemented both the `Male` and `Female` class. We'd still like to have this functionality guarantee, but in order to do it in this case, we need to somehow get around having multiple implementations. So instead of telling the users that we have the implementation right here, we say hey, we promise we'll have these methods when we're implemented...we just can't implement them yet. Think of an interface as a contract between the user of a class that implements the interface and the programmer who provides the implementation of the methods. The interface allows us to guarantee functionality AND be more extendible.

Another benefit of interfaces is that they allow you to write more generic code that reduces code duplication. For example, suppose we had some method that prints out the ages of an array of `Male`s (where class `Male` implements interface `MaleInt`):

```
public void printAges( Male[] males ) {
    for( i=0; i < males.length; i++ ) {
        System.out.println( males[i].getName() + "'s age is " +  males[i].getAge() );
    }
}
```

And now suppose we'd like to have the same method for an array of `Female`s (where class `Female` implements interface `FemaleInt`):

```
public void printAges( Female[] females ) {
    for( i=0; i < females.length; i++ ) {
        System.out.println( females[i].getName() + "'s age is " + females[i].getAge() );
    }
}
```

Gee, they look awfully similar to me. I wish I hadn't had to type that out twice, don't you?! Well HEY I have a `PersonInt` interface, right? What if I just write one method that takes in a `PersonInt`, and since both `Male` and `Female` implement this interface (since `FemaleInt` and `MaleInt` extend `PersonInt`), we could pass them as arguments:

```
public void printAges( PersonInt[] persons ) {
    for( i=0; i < persons.length; i++ ) {
        System.out.println( persons[i].getName() + "'s age is " + persons[i].getAge() );
    }
}
```

And so we've cut our code in half. In fact, we could use this printAges method on any class that implements `PersonInt`, including `Transvestite`.

7.3 Hierarchies

The hierarchies with multiple inheritance can get really complicated:

```
public interface PersonInt                                  { ... }
public interface MaleInt      extends    PersonInt          { ... }
public interface FemaleInt    extends    PersonInt          { ... }
public class     Transvestite implements MaleInt, FemaleInt { ... }
public class     Female       implements FemaleInt          { ... }
public class     Male         implements MaleInt            { ... }
```

And with more complicated hierarchies comes more complicated rules as to what polymorphism is allowed:

```
Transvestite t = new Transvestite();
Male         m = new Male();
(MaleInt)t                          // OK because t a subtype of MaleInt
(MaleInt)m                          // OK because m a subtype of MaleInt
(FemaleInt)t                        // OK because t a subtype of FemaleInt
(FemaleInt)m                        // not OK because m NOT a subtype of FemaleInt
```

8 The Iterator Interface

Here is a listing of the Java-provided Iterator interface. You'll want to familiarize yourself with it, because it's very useful, and besides, it's on your assignment:

```
// interface Iterator
// An iterator provides the mechanism to iterate through a group of
// objects one at a time.  It has the functionality to check if
// a next object exists, and if so it can retrieve it.
interface Iterator {
    boolean hasNext();      // Returns true if there is another object
    Object  next();         // Returns the next object if there is one
    void    remove();       // Ignore me, I'm never used
}
```

And here is an example of implementing the interface:

```
class Boogie implements Iterator {
    protected int[]  a;              // the int array I'll be enumerating
    protected int    cursor = 0;     // index of next elt to be enumerated

    // Constructor:  takes in the array to enumerate and sets our personal copy
    public Boogie( int a[] )         { this.a = a; }

    // Returns whether there is another elt in our array
    public boolean hasNext() {
        return ( cursor < a.length );
    }

    // Returns the next elt in our array, if we're not already at the end
    public Object next() {
        int temp = a[ cursor ];     // Temporarily store the next elt
        cursor++;                   // Advance the cursor
        return temp;                // Return the next elt
    }

    public void remove() {}         // Unimplemented
}
```

There are some problems with this iterator:

1. the user can only iterate through the elements once! Can we reset the cursor somehow? (Answer: not really)
2. making the data class implement Iterator directly is a bit of a sham because the data class' concern should only be with the data, not with the enumeration of data

To fix the second problem, we may consider making two classes: one for the data, called `Boogie`, and one to iterate over the data, called `BoogieWoogie`:

```
class Boogie {
    protected int[] a;               // The same int array over which we want to iterate
    public Boogie( int a[] )         { this.a = a; }
}

class BoogieWoogie implements Iterator {
    int    cursor = 0;
    Boogie myBoogie;

    public BoogieWoogie( Boogie b )  { myBoogie = b; }

    public boolean hasNext() {
        return ( cursor < myBoogie.a.length );
    }

    public Object next() {
        int temp = myBoogie.a[ cursor ];  // Get the next element
        cursor++;                         // Advance the cursor
        return temp;                      // Return the next element
    }

    public void remove() {}               // Unimplemented
}
```

So now the `Boogie` class focuses on data and the `BoogieWoogie` class focuses on enumeration. But, the BoogieWoogie code relies on being able to access `Boogie` variables such as array `a`. This is bad information hiding. Furthermore, `BoogieWoogie` is specialized to `Boogie`, but its code appears outside of the `Boogie` class. This is bad encapsulation. What if we changed some part of `Boogie` and forgot to tell `BoogieWoogie`? Surely the world would end.

The solution we propose to fix these problems is to declare our class `BoogieWoogie` *inside* of class `Boogie`! Then the related code would be in one class (good encapsulation) AND class `BoogieWoogie` could access the private members of `Boogie` (good information hiding)! This could work! But does Java allow it?

Lucky for us, Java allows us to declare a class within a class, in fact, there is even a name for it: inner classes.

9 Inner Classes

An *inner class* is just what it sounds like — it is a class declared inside another class. They can be defined at many levels:

1. member-level: an inner class defined as though it were just another method
2. statement-level: an inner class defined as if it were a statement in a method
3. expression-level: an inner class defined as if it were part of an expression. These are also known as *anonymous classes*.

```
class Boogie {
    private int i;                          // a private integer

    public Boogie( int arg )                { i = arg; }

    // Makes a new instance of the inner class
    public BoogieWoogie makeBoogieWoogie()  { return new BoogieWoogie(); }

    // inner class
    public class BoogieWoogie {
        public void see() {
            System.out.println( i );        // inner class can see i
        }
    }
}
```

And we can write the following code:

```
Boogie              boog     = new Boogie( 5 );
Boogie.BoogieWoogie boogwoog = boog.makeBoogieWoogie();
boogwoog.see();              // Will print out 5
```

Some things to note about inner classes:

- inner classes can be public, private, or protected
- an inner class is instantiated by a method of the containing class (as in `makeBoogieWoogie()` above) or by outerObj.new InnerClass() (as in `Boogie.new BoogieWoogie()`)
  * `new boog.BoogieWoogie()` does not work, but `Boogie.new BoogieWoogie()` does
- instances of inner classes have access to all the members of the containing outer class instance.
- the keyword `this` in `BoogieWoogie` refers to the `BoogieWoogie` object, not to the `Boogie` object. We can get around this by saving the `Boogie` object to a field member in the `Boogie` constructor.

9.1 Nested Classes

A *nested class* is a static class defined inside another class. I.e. it is a STATIC inner class. Static! That's the only difference. A nested class can only reference static (and not non-static) variables of the class in which it is nested. Why is this?

9.2 Anonymous Classes

An *anonymous class* is a class that is only going to be used once, so we don't give it a name. It is a class declaration with a usual body, but it

- is an inner class
- has no name
- has no constructor
- has no modifier (public, private, protected)
- has no explicit `extends` or `implements`

An anonymous class either extends one class or implements one interface. If it is class `A` that is extending class `P` then we write

```
new P( ... ) { body of A };
```

If it is a class `A` implementing interface `I` then we write

```
new I { body of A };
```

10 Review

Polymorphism is tricky stuff. A derived class is compatible with its base class, but not vice-versa. This makes for interesting casting issues.

The keyword `super` refers to an objects' subclass, while the keyword `this` refers to the current instance of the object. You can use both `super` and `this` as method calls, as expressions, and as objects.

An abstract class

1. cannot be directly instantiated.
2. may have some methods that are not abstract.
3. is somewhere between an interface and a concrete class
4. is part of the class hierarchy, and usual subtyping rules apply.

An interface is the extreme example of an abstract class, where no methods are implemented. A class can implement many interfaces, but it can extend only one class. Two good things about interfaces are:

1. they break code into chunks — good software engineering
2. they let you write more generic code — good code reuse

An inner class is a class defined within an outer class, a nested class is a static class defined within another class, and an anonymous class is a class defined without a name.