Lab05: Inheritance and polymorphism

Inheritance

*Inheritance* creates a new class definition by building upon an existing definition (you extend the original class)

The new class can, in turn, can serve as the basis for another class definition

- all Java objects use inheritance
- every Java object can trace back up the inheritance tree to the generic class Object

The keyword extends is used to base a new class upon an existing class

Several pairs of terms are used to discuss class relationships (these are not keywords)

![Inheritance Diagram]

- note that traditionally the arrows point from the inheriting class to the base class, and the base class is drawn at the top - in the *Unified Modeling Language (UML)* the arrows point from a class to another class that it depends upon (and the derived class depends upon the base class for its inherited code)
- the parent class/child class terms are not recommended, since parent and child is more commonly used for ownership relationships (like a GUI window is a parent to the components placed in it)

A derived class instance may be used in any place a base class instance would work - as a variable, a return value, or parameter to a method

Inheritance is used for a number of reasons (some of the following overlap)

- to model real-world hierarchies
- to have a set of pluggable items with the same "look and feel," but different internal workings
- to allow customization of a basic set of features
- when a class has been distributed and enhancements would change the way existing methods work (breaking existing code using the class)
- to provide a "common point of maintenance"

When extending a class, you can add new properties and methods, and you can change the behavior of existing methods (which is called *overriding* the methods)

- you can declare a method with the same signature and write new code for it
you can declare a property again, but this does not replace the original property - it *shadows* it (the original property exists, but any use of that name in this class and its descendants refers to the memory location of the newly declared element)

Inheritance Examples

Say you were creating an arcade game, with a number of different types of beings that might appear - wizards, trolls, ogres, princesses (or princes), frogs, etc.

- all of these entities would have some things in common, such as a name, movement, ability to add/subtract from the player's energy - this could be coded in a base class Entity
- for entities that can be chosen and controlled by a player (as opposed to those that merely appear during the course of the game but can't be chosen as a character) a new class Playable could extend Entity by adding the control features
- then, each individual type of entity would have its own special characteristics - those that can be played would extend Playable, the rest would simply extend Entity
- you could then create an array that stored Entity objects, and fill it with randomly created objects of the specific classes
- for example, your code could generate a random number between 0 and 1; if it is between 0.0 and 0.2, create a Wizard, 0.2 - 0.4 a Prince, etc.

The Java API is a set of classes that make extensive use of inheritance

- one of the classes used in the GUI is Window - its family tree looks like:
Payroll with Inheritance

Our payroll program could make use of inheritance if we had different classes of employees: exempt employees, non-exempt employees, and contract employees

- they all share basic characteristics such as getting paid (albeit via different algorithms), withholding, having to accumulate year-to-date numbers for numerous categories
- but they have different handling regarding payment calculations, benefits, dependents, etc.
- exempt employees get a monthly salary, while nonexempt get a wage * hours, contract employees are handled similarly to nonexempt, but cannot have dependents

Also, we have already seen some duplication of effort in that our dependents store some of the same information as employees (first and last names)

- they use this information for the same purposes, so it might make sense to pull that common information into a base class

This would leave us with an inheritance scheme as follows:

![Inheritance Diagram]

Note that a scheme with ContractEmployee extending NonexemptEmployee might also be a reasonable approach

Derived Class Objects

You can view a derived class object as having a complete base class object inside it

- let’s assume that the Entity class defines the properties name, energy, and position, and methods moveTo() and changeEnergy()
- the Playable class adds a property playerID
- the Wizard class adds a spells property (an array of spells they can cast) and a castSpell() method
Any Wizard object contains all the elements inside its box, include those of the base classes

- so, for example, the complete set of properties in a Wizard object is:
  - name
  - energy
  - position
  - playerID
  - spells
- a Wizard reference to a Wizard object has access to any public elements from any class in the inheritance chain from Object to Wizard
- code inside the Wizard class has access to all elements of the base classes (except those defined as private in the base class - those are present, but not directly accessible)
- a more complete description of access levels is coming up in a few pages

Note: although it appears that a base class object is physically located inside the derived class instance, it is not actually implemented that way

**Polymorphism**

**Inheritance and References**

If a derived class extends a base class, it is not only considered an instance of the derived class, but an instance of the base class as well

- the compiler knows that all the features of the base class were inherited, so they are still there to work in the derived class (keeping in mind that they may have been changed)

This demonstrates what is known as an *IsA* relationship - a derived class object *Is A* base class instance as well

- it is an example of *polymorphism* - that one reference can store several different types of objects
- for example, in the arcade game example, for any character that is used in the game, an Entity reference variable could be used, so that at runtime, any subclass can be instantiated to store in that variable

```java
Entity shrek = new Ogre();
Entity merlin = new Wizard();
```
for the player's character, a Playable variable could be used

Playable charles = new Prince();

When this is done, however, the only elements immediately available through the reference are those known to exist; that is, those elements defined in the reference type object

- the compiler decides what to allow you to do with the variable based upon the type declared for the variable
- merlin.moveTo() would be legal, since that element is guaranteed to be there
- merlin.castSpell() would not be legal, since the definition of Entity does not include it, even though the actual object referenced by w does have that capability
- the following example gives a hint as to why this is the case:

Entity x;
if (Math.random() < 0.5) x = new Wizard();
else x = new Troll();

there is no way the compiler could determine what type of object would actually be created.

- the variables names above, shrek, merlin, and charles, are probably not good choices: presumably we know shrek is an ogre, and always will be, so the type might as well be Ogre (unless, of course, he could transmogrify into something else during the game ...)

Dynamic Method Invocation

When a method is called through a reference, the JVM looks to the actual class of the instance to find the method. If it doesn't find it there, it backs up to the ancestor class (the class this class extended) and looks there (and if it doesn't find it there, it backs up again, potentially all the way to Object).

Sooner or later, it will find the method, since if it wasn't defined somewhere in the chain of inheritance, the compiler would not have allowed the class to compile.

In this manner, what you could consider the most advanced (or most derived) version of the method will run, even if you had a base class reference.

So, for our arcade game, an Entity reference could hold a Wizard, and when the move method is called, the Wizard version of move will run.

An interesting aspect of dynamic method invocation is that it occurs even if the method is called from base class code. If, for example:

- the Entity class move method called its own toString method
- the Ogre class didn't override move, but did override toString
- for an Ogre stored in an Entity variable, the move method was called

The Entity version of move would run, but its call to toString would invoke the toString method from Ogre!

Creating a Derived Class

The syntax for extending a base class to create a new class is:

Syntax
if you do not extend any class, Java assumes you are extending Object by default

Your new class can use the properties and methods contained in the original class (subject to the note coming up in a few pages about access keywords), add new data properties and methods, or replace properties or methods

A derived class object may be stored in a base class reference variable without any special treatment - the reverse is not true, but can be forced

Java doesn’t allow *multiple inheritance*, where one class inherits from two or more classes

- it does have a concept called an *interface*, which defines a set of method names
- a class may implement an interface, defining those methods in addition to whatever other methods are in the class
- this allows for several otherwise unrelated classes to have the same set of method names available, and to be treated as the same type of object for that limited set of methods

### Inheritance Example - A Derived Class

When a derived class is created, an object of the new class will in effect contain a complete object of the base class within it

The following maps out the relation between the derived class and the base class (note that the diagrams show the apparent memory allocation, in a real implementation the base class memory block is not inside the derived class block)

```java
class MyBase {
    public int x;
    public void show() {
        System.out.println("x =" + x);
    }
}
class MyDerived extends MyBase {
    public int y;
    public void show() {
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
```

Since everything in MyBase is public, code in the MyDerived class has free access to the x value from the MyBase object inside it, as well as y and show() from itself

- the show() method from MyBase is also available, but only within MyDerived class code (but some work is required to get it, since it is hidden by the show() method added with MyDerived) - code in other classes cannot invoke the MyBase version of show() at all.
Inheritance and Access

When inheritance is used to create a new (derived) class from an existing (base) class, everything in the base class is also in the derived class

- it may not be accessible, however - the access in the derived class depends on the access in the base class:

<table>
<thead>
<tr>
<th>base class access</th>
<th>accessibility in derived class</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>public</td>
</tr>
<tr>
<td>protected</td>
<td>protected</td>
</tr>
<tr>
<td>private</td>
<td>inaccessible</td>
</tr>
<tr>
<td>unspecified</td>
<td>unspecified (package access)</td>
</tr>
</tbody>
</table>

Note that private elements become inaccessible to the derived class - this does not mean that they disappear, or that there is no way to affect their values, just that they can't be referenced by name in code within the derived class

Also note that a class can extend a class from a different package

Inheritance and Constructors - the super Keyword

Since a derived class object contains the elements of a base class object, it is reasonable to want to use the base class constructor as part of the process of constructing a derived class object

- constructors are "not inherited"
- in a sense, this is a moot point, since they would have a different name in the new class, and can't be called by name under any circumstances, so, for example, when one calls new Integer(int i) they shouldn't expect a constructor named Object(int i) to run

Within a derived class constructor, however, you can use super( parameterList ) to call a base class constructor

- it must be done as the first line of a constructor
- therefore, you can't use both this() and super() in the same constructor function
- if you do not explicitly invoke a form of super-constructor, then super() (the form that takes no parameters) will run
- and for the superclass, its constructor will either explicitly or implicitly run a constructor for its superclass
- so, when an instance is created, one constructor will run at every level of the inheritance chain, all the way from Object up to the current class

Code Sample: Inheritance1.java

```java
class MyBase {
    private int x;
    public MyBase(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
    public void show() {
        System.out.println("x=" + x);
    }
}
```
class MyDerived extends MyBase {
    private int y;
    public MyDerived(int x) {
        super(x);
    }
    public MyDerived(int x, int y) {
        super(x);
        this.y = y;
    }
    public int getY() {
        return y;
    }
    public void show() {
        System.out.println("x = " + getX());
        System.out.println("y = " + y);
    }
}

public class Inheritance1 {
    public static void main(String[] args) {
        MyBase b = new MyBase(2);
        b.show();
        MyDerived d = new MyDerived(3, 4);
        d.show();
    }
}

Code Explanation

The diagram below shows the structure of our improved classes:

- a MyDerived object has two constructors available, as well as both the getX and getY methods and the show method
- both MyDerived constructors call the super constructor to handle storage of x
- the show method in the derived class overrides the base class version
- x from the base class is not available in the derived class, since it is private in MyBase, so the show method in MyDerived must call getX() to obtain the value

Derived Class Methods That Override Base Class Methods

As we saw before, you can create a method in the derived class with the same name as a base class method

- the new method overrides (and hides) the original method
- you can still call the base class method from within the derived class if necessary, by adding the super keyword and a dot in front of the method name
- the base class version of the method is not available to outside code
• you can view the super term as providing a reference to the base class object buried inside the derived class
• you cannot do super.super. to back up two levels
• you cannot change the return type when overriding a method, since this would make polymorphism impossible

Example: a revised MyDerived using super.show()

```java
class MyDerived extends MyBase {
    int y;
    public MyDerived(int x) {
        super(x);
    }
    public MyDerived(int x, int y) {
        super(x);
        this.y = y;
    }
    public int getY() {
        return y;
    }
    public void show() {
        super.show();
        System.out.println("y = " + y);
    }
}
```

**Inheritance and Default Base Class Constructors**

One base class constructor will always run when instantiating a new derived class object

• if you do not explicitly call a base class constructor, the no-arguments base constructor will be automatically run, without the need to call it as super()
• but if you do explicitly call a base class constructor, the no-arguments base constructor will not be automatically run
• the no-arguments (or no-args for short) constructor is often called the default constructor, since it the one that will run by default (and also because you are given it by default if you write no constructors).
Code Sample:

```java
class Purple {
    protected int i = 0;
    public Purple() {
        System.out.println("Purple() running and i = \" + i + ");
    }
    public Purple(int i) {
        this.i = i;
        System.out.println("Purple(i) running and i = \" + i + ");
    }
}
class Violet extends Purple {
    Violet() {
        System.out.println("Violet() running and i = \" + i + ");
    }
    Violet(int i) {
        System.out.println("Violet(i) running and i = \" + i + ");
    }
}
public class Inheritance2 {
    public static void main(String[] args) {
        new Violet();
        new Violet(4);
    }
}
```

Code Explanation

Each constructor prints a message so that we can follow the flow of execution. Note that using new Violet() causes Purple() to run, and that new Violet(4) also causes Purple() to run.

If your base class has constructors, but no no-arguments constructor, the derived class must call one of the existing constructors with super(args), since there will be no default constructor in the base class.

If the base class has a no-arguments constructor that is private, it will be there, but not be available, since private elements are hidden from the derived class. So, again, you must explicitly call an available form of base class constructor, rather than relying on the default

- try the above code with the Purple() constructor commented out or marked as private

The Instantiation Process at Runtime

In general, when an object is instantiated, an object is created for each level of the inheritance hierarchy. Each level is completed before the next level is started, and the following takes place at each level:

1. the memory block is allocated
2. the entire block is zeroed out
3. explicit initializers run - which may involve executable code, for example: private double d = Math.random();
4. the constructor for this level runs
   - since the class code has already been loaded, and any more basic code has been completed, any methods in this class or inherited from superclasses are available to be called from the constructor
   - when you call a superconstructor, all you are really doing is selecting which form of superconstructor will run - timingwise, it was run before we got to this point

When the process has completed, the expression that created the instance evaluates to the address of the block for the last unit in the chain.
Example - Factoring Person Out of Employee and Dependent

**Code Sample: Person.java**

```java
package employees;

public class Person {
    private String firstName;
    private String lastName;

    public Person() {
    }

    public Person(String firstName, String lastName) {
        setFirstName(firstName);
        setLastName(lastName);
    }

    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() { return lastName; }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getFullName() {
        return firstName + " " + lastName;
    }
}
```

**Code Explanation**

This class includes the name properties and related set and get methods.

**Code Sample: Dependent.java**

```java
package employees;

public class Dependent extends Person {
    private Employee dependentOf;

    public Dependent(Employee dependentOf, String firstName, String lastName) {
        super(firstName, lastName);
        this.dependentOf = dependentOf;
    }

    public Employee getDependentOf() { return dependentOf; }
}
```

**Code Explanation**

Since this class extends Person, the name-related elements are already present, so we remove them from this code.

**Code Sample: Employee.java**

```java
package employees;

public class Employee extends Person {
    private static int nextId = 1;
    private int id = nextId++;
```
private int dept;
private double payRate;
private Dependent[] dependents = new Dependent[5];
private int numDependents = 0;

public Employee() {
}

public Employee(String firstName, String lastName) {
    super(firstName, lastName);
}

public Employee(String firstName, String lastName, int dept) {
    super(firstName, lastName);
    setDept(dept);
}

public Employee(String firstName, String lastName, double payRate) {
    super(firstName, lastName);
    setPayRate(payRate);
}

public Employee(String firstName, String lastName, int dept, double payRate) {
    this(firstName, lastName, dept);
    setPayRate(payRate);
}

public int getId() { return id; }

public int getDept() { return dept; }

public void setDept(int dept) {
    this.dept = dept;
}

public double getPayRate() { return payRate; }

public void setPayRate(double payRate) {
    this.payRate = payRate;
}

public void addDependent(String fName, String lName) {
    if (numDependents < dependents.length) {
        dependents[numDependents++] = new Dependent(this, fName, lName);
    }
}

public String listDependents() {
    if (dependents == null) return "";
    StringBuffer temp = new StringBuffer();
    String newline = System.getProperty("line.separator");
    if (newline == null) newline = \\
    for (int i = 0; i < numDependents; i++) {
        temp.append(dependents[i].getFirstName());
        temp.append(" ");
        temp.append(dependents[i].getLastName());
        temp.append(newline);
    }
    return temp.toString();
}

public String getPayInfo() {
    return "Employee " + id + " dept " + dept + " " +
           getFullName() +
           " paid " + payRate;
}

Code Explanation

The same changes that were made to Dependent can be made here. Note that since getPayInfo
calls getFullName, which is now inherited and publicly accessible, that code did not need to change.
import employees.*;
import util.*;

public class Payroll {
    public static void main(String[] args) throws Exception {
        String fName = null;
        String lName = null;
        int dept = 0;
        double payRate = 0.0;
        double hours = 0.0;
        int numDeps = 0;
        String dfName = null;
        String dlName = null;

        Employee e = null;

        fName = JOptionPane.showMessageDialog(null,"Enter first name: ");
        lName = JOptionPane.showMessageDialog(null,"Enter last name: ");
        dept = JOptionPane.showMessageDialog(null,"Enter department: ");
        do {
            payRate = JOptionPane.showMessageDialog(null,"Enter pay rate: ");
            if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
        } while (payRate < 0.0);
        e = new Employee(fName, lName, dept, payRate);
        numDeps = JOptionPane.showMessageDialog(null,"How many dependents? ");
        for (int d = 0; d < numDeps; d++) {
            dfName = JOptionPane.showMessageDialog(null,"Enter dependent first name: ");
            dlName = JOptionPane.showMessageDialog(null,"Enter dependent last name: ");
            e.addDependent(dfName, dlName);
        }
        System.out.println(e.getPayInfo());
        System.out.println(e.listDependents());
    }
}

Code Explanation

No changes need to be made to Payroll to take advantage of the addition of the inheritance hierarchy that we added - the only changes we made were for the sake of brevity.

To revisit the sequence of events when instantiating a Dependent using the constructor that accepts the Employee and first and last names:

1. memory for an Object is allocated
2. any Object initializers would run
3. the Object() constructor would run
4. memory for a Person is allocated
5. it there were any Person initializers, they would run
6. the Person(String firstName, String lastName) constructor would run, because that was the version selected by the Dependent constructor we called
7. memory for a Dependent is allocated
8. it there were any Dependent initializers, they would run
9. the Dependent constructor we called would run
Exercise: Payroll-Inheritance01: Adding Types of Employees

We wish to improve our payroll system to take account of the three different types of employees we actually have: exempt, non-exempt, and contract employees. Rather than use some sort of identifying code as a property, OOP makes use of inheritance to handle this need, since at runtime a type can be programmatically identified.

Also, the solution code builds upon the Person base class from the preceding example. You can either copy the Person.java file into your working directory and edit Employee.java to match.

1. Create three new classes: ExemptEmployee, NonexemptEmployee, and ContractEmployee that extend Employee.

2. In our company, exempt employees get a monthly salary, non-exempt an hourly rate that is multiplied by their hours, as do contract employees - revise the `getPayInfo` method to take this into account and also identify which type of employee it is (note that the `payRate` field will hold a monthly amount for exempt and an hourly amount for non-exempt and contractors, but these last two will need an additional field for their hours, plus methods to set and get the hours)

3. We need some way to handle the fact that contract employees do not have dependents - for now, just override listDependents to do nothing for a contract employee, and also override the addDependent method to do nothing for a ContractEmployee
   o we can see here a problem with the hard-coded instantiation of the dependents array in Employee
   o a better approach might be to leave it as null, and add a method to create the array, which could accept a size at that time (we won't bother to do this, since the collections classes offer a much better way that we will implement later)

4. Add constructors as you see fit

5. Have the main program create and display the information for instances of each type of employee (for now, comment out the code that uses the Employee array and just hard-code one of each type of employee).