**Day 30 – Inheritance II**

**Topic – Overriding Methods**

**Overloaded** methods, as we have seen, have the same name as one another, but differ in the number and types of their parameters. With **overridden** methods, we replace a method in the super class with a new version of it in the subclass. For this to be overriding:

* Both methods must have the same name
* Both methods must have the same # of parameters
* The parameters for both classes must have the same type

A method in a subclass can **overload** a method in a super class, but for it to be overloading, the number and types of the parameters must differ.

**Example:** In today’s downloads there is a super class, called Employee, and two subclasses, called HourlyEmployee and SalariedEmployee. Open these up and take a look at each.

Compile each and run TestEmployee.java

**NOTE** the following:

In the main program… emp1 **is-an** Employee.   
 Also… emp2 **is-an** Employee.

In SalariedEmployee AND HourlyEmployee, there are not methods called   
 getFirstName or getLastName  
 although **both**are called in the main program (TestEmployee).  
 Similarly for setFirstName and setLastName.  
 This is because these are **inherited** from the Employee super class.  
  
 The toString methods in Salaried and Hourly Employee **override** the toString   
 method in Employee.|  
  
 These methods also call the toString method of Employee to print out Employee  
 attributes, using the syntax super.toString() which means to call the version of  
 toString inherited from the Employee super class.  
  
 The printStub method in Salaried and Hourly Employee each access the firstName and  
 lastName attributes of Employee. They do this via the accessors in  
 Employee. They cannot directly access the attributes because the attributes  
 are **private**.

In TestEmployee, the line:  
 SalariedEmployee emp1 = new SalariedEmployee(  
 "George", "Johnson", 86700.00)  
 Creates the following items in memory and sets emp1 to be a reference to them:  
 firstName "George"  
 lastName "Johnson"  
 yearlySalary 86700.00  
  
 while the line:  
 HourlyEmployee emp2 = new HourlyEmployee(  
 "Martha", "Bloomington", 45.25);  
   
 Creates the following items in memory and sets emp2 to be a reference to them:  
 firstName "Martha"   
 lastName "Bloomington"  
 payRate 45.25

**NO** **overloading** takes place here, since there are no new (additional) methods in either  
SalariedEmployee or HourlyEmployee that are the same name, with different parameters, than other methods in those two classes OR in Employee.

**Topic – Casting and Inheritance**

**Upcasting**Any sub class (e.g., HourlyEmployee or SalariedEmployee) (or sub-sub class or sub-sub-sub class, etc.) can be cast as any of its super classes (e.g., Employee). This is an example of **upcasting** (like int to double) which is allowed since no information is lost. Everything in a super class is also in the sub class.

So, if we have:  
 HourlyEmployee emp1 = new HourlyEmployee( … );  
 SalariedEmployee emp2 = new SalariedEmployee( … );  
 Employee emp3;  
we can write:  
 emp3 = emp1;  
OR  
 emp3 = emp2;

**Downcasting**Converting to a super class (e.g., Employee) has a weird result. If we do this it would seem we lose the additional attributes or methods of the subclass. For example:  
 Employee emp4 = new HourlyEmployee("Jeff", "Baldwin", 35.67);  
and  
 Employee emp5 = new SalariedEmployee("Joseph", "WingDing", 35678.00);  
are allowed, but then we cannot call:  
 emp4.getWeeklyPay(40.0);  
 or  
 emp5.getYTDPay(11);  
Since emp4 and emp5 are of class Employee which does not include these methods.   
  
Java, however, ‘knows’ that emp4 is really an HourlyEmployee and emp5 is really a SalariedEmployee and you can access these subclass methods by explicit downcasting:  
 ((HourlyEmployee)emp4).getWeeklyPay(40.0);  
and  
 ((SalariedEmployee)emp5).getYTDPay(11);  
are allowed.

**instanceof**You cannot downcast an object to a class it does not really belong to. For example:  
 ((SalariedEmployee)emp4).getWeeklyPay(40.0);  
will result in a Class Cast Exception (a runtime error) because emp4 was not created as an HourlyEmployee.

To avoid such errors, you should check if they are allowed before casting:  
 if(emp4 **instanceof** SalariedEmployee) {  
 ((SalariedEmployee)emp4).getWeeklyPay(40.0);  
 }

This **instanceof** operator can be thought of as an **is-a(n)** operator … that is, was the given object created as the given class??

**IN CLASS EXERCISE**Using Employee, SalariedEmployee, and HourlyEmployee … write a main class (TestEmployee2.java) that instantiates an ArrayList of Employee objects:  
 ArrayList<Employee> myList = new ArrayList<Employee>();

Then, in a loop, manage the following conversation:  
 ----jGRASP exec: java TestEmployee2  
First Name: Judy  
Last Name: Brown  
Salaried or Hourly (S/H): S  
Salary: 34500.00  
First Name: Fred  
Last Name: Flintstone  
Salaried or Hourly (S/H): H  
Hourly Rate: 25.50  
First Name: Barney  
Last Name: Rubble  
Salaried or Hourly (S/H): S  
Salary: 65780.00  
First Name: Mary  
Last Name: Qos  
Salaried or Hourly (S/H): H  
Hourly Rate: 45.67  
First Name:   
Salaried: Judy Brown Salary: 34500.0 Pay per period: 1437.50  
Hourly: Fred Flintstone Hourly Rate: 25.5  
Salaried: Barney Rubble Salary: 65780.0 Pay per period: 2740.83  
Hourly: Mary Qos Hourly Rate: 45.67  
  
 ----jGRASP: operation complete.

The idea here is that we are entering employee data. At the end, we print it out, just to check.

First we enter the employee’s name. If the first name is the empty string (just an ENTER) we are done entering employee data. Otherwise, we continue by entering their category (hourly or salaried). This is an H or and S entered by the user.

If salaried, we enter their salary. Then, create a SalariedEmployee object and add it  
 (myList.add(…))to the list of employees. The list is a list of Employee objects, but   
 that’s OK because of automatic upcasting.  
  
 if hourly, we enter their hourly pay rate. Then, create an HourlyEmployee object and   
 add it (myList.add(…)) to the list of employees. Again, automatic upcasting allows this.

When all employee data has been entered (the answer for the first name is the empty string) we print out all employee data. We can use a regular for loop for this:  
 for(int i = 0; i < myList.size(); i++) { … }  
or we can use an enhanced forloop for this:  
 for(Employee e : myList) { … }  
which executes the body of the loop ({ … }) once for each employee, e.

In the loop, we use instanceof to determine if the employee is a SalariedEmployee or an HourlyEmployee … then, we downcast the employee (e) to that type and print the result of calling toString(). The downcast ensures that the appropriate version of toString is called.