CMSC 202H

Inheritance I

Class Reuse with Inheritance

Class Reuse

- We have seen how classes (and their code) can be reused with composition.
  - An object has another object as one (or more) of its instance variables.
- Composition models the “has a” relationship.
  - A Person has a String (name)
  - A Car has an Engine
  - A Book has an array of Pages

Object Relationships

- An object can be a specialized version of another object.
  - A Car is a Vehicle
  - A Triangle is a Shape
  - A Doctor is a Person
  - A Student is a Person

  This kind of relationship is known as the “is a type of” relationship.
- In OOP, this relationship is modeled with the programming technique known as **inheritance**.
- Inheritance creates new classes by adding code to an existing class. The existing class is reused without modification.

Introduction to Inheritance

- *Inheritance* is one of the main techniques of OOP.

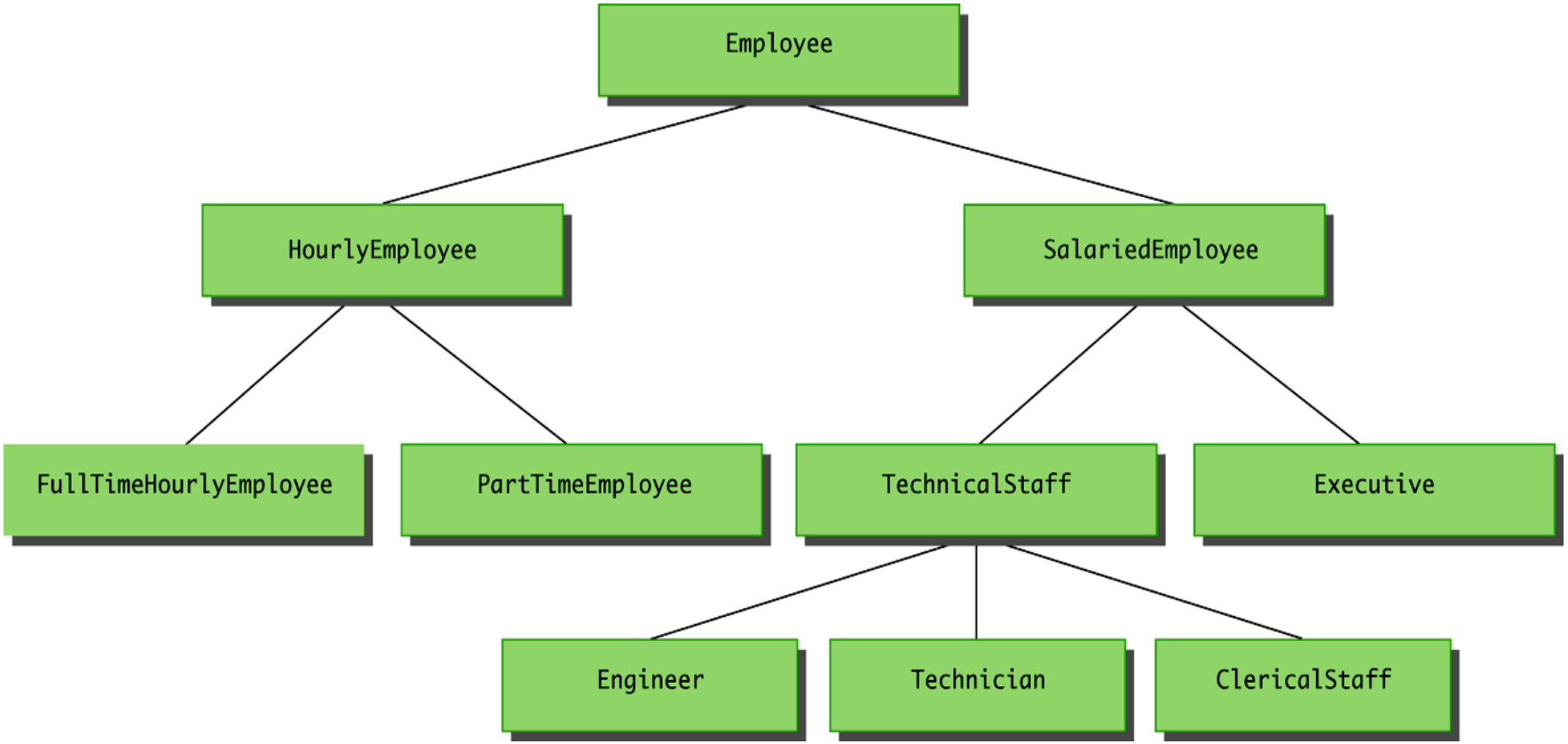
- Using inheritance
  - a very general class is first defined,
  - then more specialized versions of the class are defined by
    - adding instance variables and/or
    - adding methods.
  - The specialized classes are said to *inherit* the methods and instance variables of the general class.

Derived Classes

- There is often a natural hierarchy when designing certain classes.
- Example:
  - In a record-keeping program for the employees of a company, there are hourly employees and salaried employees.
  - Hourly employees can be further divided into full time and part time workers.
  - Salaried employees can be divided into those on the technical staff and those on the executive staff.

A Class Hierarchy

Display 7.1 **A Class Hierarchy**



Derived Classes

- All employees have certain characteristics in common:
  - a name and a hire date
  - the methods for setting and changing the names and hire dates
- Some employees have specialized characteristics:
  - Pay
    - hourly employees are paid an hourly wage
    - salaried employees are paid a fixed wage
  - Calculating wages for these two different groups would be different.

Inheritance and OOP

- Inheritance is an abstraction for
  - sharing similarities among classes (name and hireDate), and
  - preserving their differences (how they get paid).
- Inheritance allows us to group classes into families of related types (Employees), allowing for the sharing of common operations and data.

General Classes

- A class called **`Employee`** can be defined that includes all employees.
  - This class can then be used as a foundation to define classes for hourly employees and salaried employees.
    - The **`HourlyEmployee`** class can be used to define a **`PartTimeHourlyEmployee`** class, and so forth.

The Employee Class

```
/**
 Class Invariant: All objects have a name string and hire date.
 A name string of "No name" indicates no real name specified yet.
 A hire date of Jan 1, 1000 indicates no real hire date specified yet.
*/
public class Employee
{
    private String name;
    private Date hireDate;

   // no-argument constructor
    public Employee( )
    {
         name = "No name";
         hireDate = new Date("Jan", 1, 1000); //Just a placeholder.
    }
   // alternate constructor
    public Employee(String theName, Date theDate)   { /* code here */ }

   // copy constructor
    public Employee(Employee originalObject)        { /* code here */ }
```

(continued)

Employee Class

```
    // some accessors and mutators
     public String getName( )                          { /* code here */ }
     public Date getHireDate( )                        { /* code here */ }
     public void setName(String newName)               { /* code here */ }
     public void setHireDate(Date newDate)             { /* code here */ }

    // everyone gets the same raise
    public double calcRaise( )
         { return 200.00; }

    // toString and equals
     public String toString( )                         { /* code here */ }
     public boolean equals(Employee otherEmployee)
         { /* code here */ }

}   // end of Employee Class
```

Derived Classes

- Since an hourly employee "**is an**" employee, we want our class **`HourlyEmployee`** to be defined as a **derived** class of the class **`Employee`**.
  - A derived class is defined by adding instance variables and/or methods to an existing class.
  - The class that the derived class is built upon is called the **base class**.
  - The phrase **`extends BaseClass`** must be added to the derived class definition:

```
public class HourlyEmployee extends Employee
```

- In OOP, a base class/derived class relationship is alternatively referred to by the term pairs:
  - **superclass/subclass**
  - **parent class/child class**

HourlyEmployee Class

```
/**
 Class Invariant: All objects have a name string, hire date,
nonnegative  wage rate, and nonnegative number of hours worked. */

public class HourlyEmployee extends Employee
{
   // instance variables unique to HourlyEmployee
    private double wageRate;
    private double hours; //for the month

   // no-argument Constructor
   public HourlyEmployee( )                                  { /* code here */}

   // alternative constructor
   public HourlyEmployee(String theName, Date theDate,
      double theWageRate, double theHours)                   { /* code here */}

   // copy constructor
    public HourlyEmployee(HourlyEmployee originalHE)         { /* code here */}
```

(continued)

HourlyEmployee Class

```
   // accessors and mutator specific to HourlyEmployee

   public double getRate( )                        { /* code here */ }
   public double getHours( )                       { /* code here */ }
   public void setHours(double hoursWorked)        { /* code here */ }
   public void setRate(double newWageRate)         { /* code here */ }

   // toString and equals specific for HourlyEmployee
   public String toString( )                       { /* code here */ }
   public boolean
        equals(HourlyEmployee otherHE)  { /* code here */ }

}  // end of HourlyEmployee Class
```

Derived Class (Subclass)

- The derived class **inherits** all of the
  - public methods (and private methods, indirectly),
  - public and private instance variables, and
  - public and private static variables

  from the base class.

Inherited Members

- The derived class **inherits** all of the
  - public methods (and private methods, indirectly),
  - public and private instance variables, and
  - public and private static variables

  from the base class.

- Definitions for the inherited variables and methods do not appear in the derived class's definition.
  - The code is reused without having to explicitly copy it, unless the creator of the derived class redefines one or more of the base class methods.

- All instance variables, static variables, and/or methods defined directly in the derived class's definition are *added* to those inherited from the base class

Using HourlyEmployee

```
public class HourlyEmployeeExample
{
    public static void main(String[] args)
    {
       HourlyEmployee joe =
       new HourlyEmployee("Joe Worker", new Date(1, 1, 2004), 50.50, 160);

       // getName is defined in Employee
       System.out.println("joe's name is " + joe.getName( ));

       // setName is defined in Employee
       System.out.println("Changing joe's name to Josephine.");
       joe.setName("Josephine");

       // setRate is specific for HourlyEmployee
       System.out.println("Giving Josephine a raise");
       joe.setRate( 65.00 );

       // calcRaise is defined in Employee
       double raise = joe.calcRaise( );
       System.out.println("Joe's raise is " + raise );
    }
}
```

Overriding a Method Definition

- A derived class can change or **override** an inherited method.
- In order to override an inherited method, a new method definition is placed in the derived class definition.
- For example, perhaps the HourlyEmployee class had its own way to calculate raises. It could override Employee's calcRaise( ) method by defining its own.

Overriding Example

```
public class Employee
{
    ....
    public double calcRaise( ) { return 200.00; }
}

public class HourlyEmployee extends Employee
{
    . . . .
    // overriding calcRaise – same signature as in Employee
    public double calcRaise( ) return 500.00; }
}
```

Now, this code

```
HourlyEmployee joe = new HourlyEmployee( );
double raise = joe.calcRaise( );
```

invokes the **overridden** `calcRaise` method in the HourlyEmployee class rather than the `calcRaise( )` method in the Employee class

To override a method in the derived class, the overriding method must have the same method signature as the base class method.

Overriding Versus Overloading

- Do not confuse *overriding* a method in a derived class with *overloading* a method name.
  - When a method in a derived class has the *same signature* as the method in the base class, that is overriding.
  - When a method in a derived class or the same class has a *different signature* from the method in the base class or the same class, that is overloading.
  - Note that when the derived class overrides or overloads the original method, it still inherits the original method from the base class as well (we'll see this later).

The `final` Modifier

- If the modifier **`final`** is placed before the definition of a *method*, then that method may not be overridden in a derived class.

- It the modifier **`final`** is placed before the definition of a *class*, then that class may not be used as a base class to derive other classes.

Pitfall: Use of Private Instance Variables from a Base Class

- An instance variable that is private in a base class is not accessible *by name* in a method definition of a derived class.
  - An object of the **`HourlyEmployee`** class cannot access the private instance variable **`hireDate`** by name, even though it is inherited from the **`Employee`** base class.
- Instead, a private instance variable of the base class can only be accessed by the public accessor and mutator methods defined in that class.
  - An object of the **`HourlyEmployee`** class can use the **`getHireDate`** or **`setHireDate`** methods to access **`hireDate`**.

Encapsulation and Inheritance Pitfall: Use of Private Instance Variables from a Base Class

- If private instance variables of a class were accessible in method definitions of a derived class, …
  - then anytime someone wanted to access a private instance variable, they would only need to create a derived class, and access the variables in a method of that class.
- This would allow private instance variables to be changed by mistake or in inappropriate ways.

Pitfall: Private Methods Are Effectively Not Inherited

- The private methods of the base class are like private variables in terms of not being directly available.
- A private method is completely unavailable, unless invoked indirectly.
  - This is possible only if an object of a derived class invokes a public method of the base class that happens to invoke the private method.
- This should not be a problem because private methods should be used only as helper methods.
  - If a method is not just a helper method, then it should be public.