Advanced Programming

Introduction to Inheritance
Introduction to Inheritance

- *Inheritance* is one of the main techniques of object-oriented programming (OOP)
- Using this technique, a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods
  - The specialized classes are said to *inherit* the methods and instance variables of the general class
Introduction to Inheritance

• Inheritance is the process by which a new class is created from another class
  – The new class is called a derived class
  – The original class is called the base class

• A derived class automatically has all the instance variables and methods that the base class has, and it can have additional methods and/or instance variables as well

• Inheritance is especially advantageous because it allows code to be reused, without having to copy it into the definitions of the derived classes
Derived Classes

• When designing certain classes, there is often a natural hierarchy for grouping them
  – In a record-keeping program for the employees of a company, there are hourly employees and salaried employees
  – Hourly employees can be divided into full time and part time workers
  – Salaried employees can be divided into those on technical staff, and those on the executive staff
Derived Classes

• All employees share certain characteristics in common
  – All employees have a name and a hire date
  – The methods for setting and changing names and hire dates would be the same for all employees
• Some employees have specialized characteristics
  – Hourly employees are paid an hourly wage, while salaried employees are paid a fixed wage
  – The methods for calculating wages for these two different groups would be different
Derived Classes

• Within Java, a class called Employee can be defined that includes all employees

• This class can then be used to define classes for hourly employees and salaried employees
  – In turn, the HourlyEmployee class can be used to define a PartTimeHourlyEmployee class, and so forth
A Class Hierarchy

Display 7.1  A Class Hierarchy

- Employee
  - HourlyEmployee
    - FullTimeHourlyEmployee
  - SalariedEmployee
    - TechnicalStaff
      - Engineer
  - PartTimeEmployee
  - Technician
  - Executive
  - ClericalStaff
/*
   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 January 1, 1000 indicates no real hire date specified yet.
*/

public class Employee
{
    private String name;
    private Date hireDate;

    public Employee()
    {
        name = "No name";
        hireDate = new Date("January", 1, 1000); // Just a placeholder.
    }

    /**
     * Precondition: Neither theName nor theDate is null.
     */
    public Employee(String theName, Date theDate)
    {
        if (theName == null || theDate == null)
        {
            System.out.println("Fatal Error creating employee.");
            System.exit(0);
        }
        name = theName;
        hireDate = new Date(theDate);
    }

    // The class Date is defined in Display 4.13.
public Employee(Employee originalObject) {
    name = originalObject.name;
    hireDate = new Date(originalObject.hireDate);
}

public String getName() {
    return name;
}

date getHireDate() {
    return new Date(hireDate);
}

/**
 * Precondition newName is not null.
 */
public void setName(String newName) {
    if (newName == null) {
        System.out.println("Fatal Error setting employee name.");
        System.exit(0);
    } else{
        name = newName;
    }
}
/**
  * Precondition newDate is not null.
  */
  
  public void setHireDate(Date newDate)
  {
      if (newDate == null)
      {
          System.out.println("Fatal Error setting employee hire date.");
          System.exit(0);
      }
      else
      {
          hireDate = new Date(newDate);
      }
  }

  public String toString()
  {
      return (name + " " + hireDate.toString());
  }

  public boolean equals(Employee otherEmployee)
  {
      return (name.equals(otherEmployee.name)
      && hireDate.equals(otherEmployee.hireDate));
  }
Derived Classes

- Since an hourly employee is an employee, it is defined as a *derived* class of the class `Employee`
  - A *derived class* is defined by adding instance variables and methods to an existing class
  - The existing 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:
    ```java
    public class HourlyEmployee extends Employee
    ```
Derived Classes

- When a derived class is defined, it is said to inherit the instance variables and methods of the base class that it extends
  - Class `Employee` defines the instance variables `name` and `hireDate` in its class definition
  - Class `HourlyEmployee` also has these instance variables, but they are not specified in its class definition
  - Class `HourlyEmployee` has additional instance variables `wageRate` and `hours` that are specified in its class definition
Derived Classes

• Just as it inherits the instance variables of the class **Employee**, the class **HourlyEmployee** inherits all of its methods as well
  – The class **HourlyEmployee** inherits the methods **getName**, **getHireDate**, **setName**, and **setHireDate** from the class **Employee**
  – Any object of the class **HourlyEmployee** can invoke one of these methods, just like any other method
Derived Class (Subclass)

• A derived class, also called a *subclass*, is defined by starting with another already defined class, called a *base class* or *superclass*, and adding (and/or changing) methods, instance variables, and static variables
  – The derived class inherits all the public methods, all the public and private instance variables, and all the public and private static variables from the base class
  – The derived class can add more instance variables, static variables, and/or methods
Class HourlyEmployee

1 /*
2  **
3  * Class Invariant: All objects have a name string, hire date, wage rate, and nonnegative number of hours worked. A name string "No name" indicates no real name specified yet. A hire date of indicates no real hire date specified yet.
4  */
5 public class HourlyEmployee extends Employee
6 {
7      private double wageRate;
8      private double hours; //for the month
9
10     public HourlyEmployee()
11     {
12         super();
13         wageRate = 0;
14         hours = 0;
15     }
16
17     /*
18      **
19      * Precondition: Neither theName nor theDate is null; theWageRate and theHours are nonnegative.
20      */
public HourlyEmployee(String theName, Date theDate, double theWageRate, double theHours)
{
    super(theName, theDate);
    if ((theWageRate >= 0) && (theHours >= 0))
    {
        wageRate = theWageRate;
        hours = theHours;
    }
    else
    {
        System.out.println("Fatal Error: creating an illegal hourly employee.");
        System.exit(0);
    }
}

public HourlyEmployee(HourlyEmployee originalObject)
{
    super(originalObject);
    wageRate = originalObject.wageRate;
    hours = originalObject.hours;
}
public double getRate()
{
    return wageRate;
}

public double getHours()
{
    return hours;
}

/**
 * Returns the pay for the month.
 */
public double getPay()
{
    return wageRate*hours;
}

/**
 * Precondition: hoursWorked is nonnegative.
 */
public void setHours(double hoursWorked)
{
    if (hoursWorked >= 0)
        hours = hoursWorked;
else {
    System.out.println("Fatal Error: Negative hours worked.");
    System.exit(0);
}

/**
 * Precondition: newWageRate is nonnegative.
 */
public void setRate(double newWageRate) {
    if (newWageRate >= 0) {
        wageRate = newWageRate;
    } else {
        System.out.println("Fatal Error: Negative wage rate.");
        System.exit(0);
    }
}
The method toString is overridden so it is different in the derived class HourlyEmployee than it is in the base class Employee.

```java
public String toString()
{
    return (getName() + " " + getHireDate().toString()
            + "\n\$" + wageRate + " per hour for " + hours + " hours");
}
```

```java
public boolean equals(HourlyEmployee other)
{
    return (getName().equals(other.getName())
            && getHireDate().equals(other.getHireDate())
            && wageRate == other.wageRate
            && hours == other.hours);
}
```

We will show you a better way to define equals later in this chapter.
Display 7.4  Inheritance Demonstration

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

        System.out.println("joe's longer name is " + joe.getName());

        System.out.println("Changing joe's name to Josephine.");
        joe.setName("Josephine");

        System.out.println("joe's record is as follows:");
        System.out.println(joe);
    }
}
```

Sample Dialogue

joe's longer name is Joe Worker
Changing joe's name to Josephine.
joe's record is as follows:
Josephine January 1, 2004
$50.5 per hour for 160 hours

The methods getName and setName are inherited from the base class Employee.
public class SalariedEmployee extends Employee
{
    private double salary; // annual

    public SalariedEmployee()
    {
        super();
        salary = 0;
    }

    public SalariedEmployee(String theName, Date theDate, double theSalary)
    {
        super(theName, theDate);
        if (theSalary >= 0)
            salary = theSalary;
        else
        {
            System.out.println("Fatal Error: Negative salary.");
            System.exit(0);
        }
    }
}
public SalariedEmployee(SalariedEmployee originalObject)
{
    super(originalObject);
    salary = originalObject.salary;
}

public double getSalary()
{
    return salary;
}

/**
 * Returns the pay for the month.
 */
public double getPay()
{
    return salary/12;
}

/**
 * Precondition: newSalary is nonnegative.
 */
public void setSalary(double newSalary)
{
    if (newSalary >= 0)
        salary = newSalary;
else
{
    System.out.println("Fatal Error: Negative salary.");
    System.exit(0);
}

public String toString()
{
    return (getName() + " " + getHireDate().toString()
            + "\n$" + salary + " per year");
}

public boolean equals(SalariedEmployee other)
{
    return (getName()).equals(other.getName())
    && getHireDate().equals(other.getHireDate())
    && salary == other.salary);  
}
Inherited Members

• A derived class automatically has all the instance variables, all the static variables, and all the public methods of the base class
  – Members from the base class are said to be *inherited*

• Definitions for the inherited variables and methods do not appear in the derived class
  – 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
Parent and Child Classes

• A base class is often called the *parent class*
  – A derived class is then called a *child class*

• These relationships are often extended such that a class that is a parent of a parent . . . of another class is called an *ancestor class*
  – If class \( A \) is an ancestor of class \( B \), then class \( B \) can be called a *descendant* of class \( A \)
Overriding a Method Definition

• Although a derived class inherits methods from the base class, it can change or **override** an inherited method if necessary
  – In order to override a method definition, a new definition of the method is simply placed in the class definition, just like any other method that is added to the derived class
  – Examples: `toString`, `equals` methods in `HourlyEmployee`
Changing the Return Type of an Overridden Method

• Ordinarily, the type returned may not be changed when overriding a method
• However, if it is a class type, then the returned type may be changed to that of any descendant class of the returned type
• This is known as a covariant return type
  – Covariant return types are new in Java 5.0; they are not allowed in earlier versions of Java
Covariant Return Type

• Given the following base class:

```
public class BaseClass
{
    public Employee getSomeone(int someKey)
    {
    }
}
```

• The following is allowed in Java 5.0:

```
public class DerivedClass extends BaseClass
{
    public HourlyEmployee getSomeone(int someKey)
    {
    }
}
```
Changing the Access Permission of an Overridden Method

• The access permission of an overridden method can be changed from private in the base class to public (or some other more permissive access) in the derived class

• However, the access permission of an overridden method can not be changed from public in the base class to a more restricted access permission in the derived class
Changing the Access Permission of an Overridden Method

• Given the following method header in a base case:
  \texttt{private void doSomething()}
• The following method header is valid in a derived class:
  \texttt{public void doSomething()}
• However, the opposite is not valid
• Given the following method header in a base case:
  \texttt{public void doSomething()}
• The following method header is \textit{not} valid in a derived class:
  \texttt{private void doSomething()}
Pitfall: Overriding Versus Overloading

• Do not confuse *overriding* a method in a derived class with *overloading* a method name
  – When a method is overridden, the new method definition given in the derived class has the exact same number and types of parameters as in the base class
  – When a method in a derived class has a different signature from the method in the base class, that is overloading
  – Note that when the derived class overloads the original method, it still inherits the original method from the base class as well
Suppose we added following to class `HourlyEmployee`:

```java
public void setName(String firstName, String lastName) {
    if ((firstName == null) || (lastName == null)) {
        System.out.println("Fatal Error setting employee name.");
        System.exit(0);
    } else {
        name = firstName + " " + lastName;
    }
}
```

Class `HourlyEmployee` would than have 2 different `setName` methods, above one and the one from class `Employee`:

```java
public void setName(String newName) {
    if (newName == null) {
        System.out.println("Fatal Error setting employee name.");
        System.exit(0);
    } else {
        name = newName;
    }
}
```
Both *HourlyEmployee* and *Employee* have `toString` method with the same heading:

```java
public String toString()
{
    return (name + " " + hireDate.toString());
}
```

```java
public String toString()
{
    return (getName() + " " + getHireDate().toString()
            + "\n$" + wageRate + " per hour for " + hours + " hours");
}
```
The **final** Modifier

- If the modifier **final** is placed before the definition of a *method*, then that method may not be redefined in a derived class.
- If 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.
The **super** Constructor

- A derived class uses a constructor from the base class to initialize all the data inherited from the base class
  - In order to invoke a constructor from the base class, it uses a special syntax:

```java
public derivedClass(int p1, int p2, double p3)
{
    super(p1, p2);
    instanceVariable = p3;
}
```
  - In the above example, `super(p1, p2);` is a call to the base class constructor
public HourlyEmployee(String theName, Date theDate,
                     double theWageRate, double theHours)
{
    super(theName, theDate);
    if (((theWageRate >= 0) && (theHours >= 0))
    {
        wageRate = theWageRate;
        hours = theHours;
    }
    else
    ...

    Employee(theName, theDate); //ILLEGAL
The **super** Constructor

- A call to the base class constructor can never use the name of the base class, but uses the keyword `super` instead.
- A call to `super` must always be the first action taken in a constructor definition.
- An instance variable cannot be used as an argument to `super`.
The `super` Constructor

• If a derived class constructor does not include an invocation of `super`, then the no-argument constructor of the base class will automatically be invoked
  – This can result in an error if the base class has not defined a no-argument constructor

• Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to `super` should always be used
public HourlyEmployee()
{
    wageRate = 0;
    hours = 0;
}

public HourlyEmployee()
{
    super();
    wageRate = 0;
    hours = 0;
}

Equivalent constructors
The **this** Constructor

- Within the definition of a constructor for a class, **this** can be used as a name for invoking another constructor in the same class
  - The same restrictions on how to use a call to **super** apply to the **this** constructor

- If it is necessary to include a call to both **super** and **this**, the call using **this** must be made first, and then the constructor that is called must call **super** as its first action
The **this** Constructor

- Often, a no-argument constructor uses **this** to invoke an explicit-value constructor
  - No-argument constructor (invokes explicit-value constructor using **this** and default arguments):
    ```java
    public ClassName()
    {
        this(argument1, argument2);
    }
    ```
  - Explicit-value constructor (receives default values):
    ```java
    public ClassName(type1 param1, type2 param2)
    {
        . . .
    }
    ```
The **this** Constructor

```java
public HourlyEmployee()
{
    this("No name", new Date(), 0, 0);
}
```

- The above constructor will cause the constructor with the following heading to be invoked:

  ```java
  public HourlyEmployee(String theName,
      Date theDate, double theWageRate, double theHours)
  ```
public HourlyEmployeee()
{
    wageRate = 0;
    hours = 0;
}

public HourlyEmployeee()
{
    super();
    wageRate = 0;
    hours = 0;
}

public HourlyEmployeee()
{
    this("No name", new Date("January", 1, 10000), 0, 0);
}
Tip: An Object of a Derived Class Has More than One Type

- An object of a derived class has the type of the derived class, and it also has the type of the base class
- More generally, an object of a derived class has the type of every one of its ancestor classes
  - Therefore, an object of a derived class can be assigned to a variable of any ancestor type
Tip: An Object of a Derived Class Has More than One Type

- An object of a derived class can be plugged in as a parameter in place of any of its ancestor classes.
- In fact, a derived class object can be used anywhere that an object of any of its ancestor types can be used.
- Note, however, that this relationship does not go the other way.
  - An ancestor type can never be used in place of one of its derived types.
public class IsADemo {

    public static void main(String[] args) {
        SalariedEmployee joe = new SalariedEmployee("Josephine",
                                                       new Date("January", 1, 2004), 100000);
        HourlyEmployee sam = new HourlyEmployee("Sam",
                                                      new Date("February", 1, 2003), 50.50, 40);

        System.out.println("joe's longer name is " + joe.getName());

        System.out.println("showEmployee(joe) invoked:");
        showEmployee(joe);

        System.out.println("showEmployee(sam) invoked:");
        showEmployee(sam);

    }

    public static void showEmployee(Employee employeeObject) {

        System.out.println(employeeObject.getName());
        System.out.println(employeeObject.getHireDate());
    }

Sample Dialogue

joe's longer name is Josephine
showEmployee(joe) invoked:
Josephine
January 1, 2004
showEmployee(sam) invoked:
Sam
February 1, 2003
Pitfall: The Terms "Subclass" and "Superclass"

• The terms *subclass* and *superclass* are sometimes mistakenly reversed
  – A superclass or base class is more general and inclusive, but less complex
  – A subclass or derived class is more specialized, less inclusive, and more complex
    • As more instance variables and methods are added, the number of objects that can satisfy the class definition becomes more restricted
Encapsulation and Inheritance Pitfall: Use of Private Instance Variables from the Base Class

• An instance variable that is private in a base class is not accessible by name in the definition of a method in any other class, not even in a method definition of a derived class
  – For example, 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
public String toString()
{
    return (getName() + " " + getHireDate().toString()
    + "\n$" + wageRate + " per hour for " + hours + " hours");
}

public String toString()  //Illegal version
{
    return (name + " " + hireDate.toString()
    + "\n$" + wageRate + " per hour for " + hours + " hours");
}
Encapsulation and Inheritance Pitfall: Use of Private Instance Variables from the 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 it in a method of that class
  – This would allow private instance variables to be changed by mistake or in inappropriate ways (for example, by not using the base type's accessor and mutator methods only)
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

• However, 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 just be used as helping methods
  – If a method is not just a helping method, then it should be public, not private
Source

• Absolute Java, Savitch
  – Copyright © 2010 Pearson Addison-Wesley. All rights reserved.