

Extending Classes

A Gregorian calendar is a type of calendar. A "calendar" is a fairly general concept, whereas a "Gregorian calendar" — the calendar used in much of the world — is a specific type of calendar. When an "is a" (or "is an") relationship exists between two entities, there is an opportunity for one entity to extend the other. For an example, let's repeat the first sentence in this paragraph with emphasis added: *A Gregorian calendar is a type of calendar.* The more specific entity inherits properties from the more general one, plus adds features unique to it. So a Gregorian calendar has all the basic characteristics of a calendar, plus some unique ones, such as rules for leap years. In Java and other object-oriented programming languages, this relationship is known as *inheritance*. The GregorianCalendar class is said to inherit from, or *extend*, the Calendar class. As we know, all Java classes inherit from, or extend, the Object class. Figure 1 illustrates this relationship for these three classes.

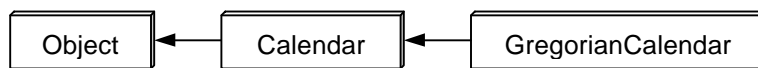


Figure 1. Inheritance relationship between the GregorianCalendar, Calendar, and Object classes

In an inheritance relationship, the more general class is the *superclass* and the more specific class is the *subclass*. So, the Object class is the superclass to the Calendar class, which, in turn, is the superclass to the GregorianCalendar class. It is correct to say that the Object class is the superclass to all classes in Java, and that all classes are subclasses of the Object class. The GregorianCalendar class is a subclass of the Object class because, by inheriting properties from the Calendar class, it implicitly inherits properties from the Object class.

Of course there are many examples in real life. Musical instruments come in many shapes and sizes. There are "general" characteristics held by all musical instruments (They all make music!), and families of instruments inherit these, plus add their own "specific" characteristics. For example, woodwind instruments all use a wooden reed, whereas brass instruments use a round metal mouthpiece. Within these families, there are specific kinds of instruments with their own features, and so on. Figure 2 shows a portion of the inheritance relationship for musical instruments. (Take a minute to think about the "is as" relationships in the figure.)

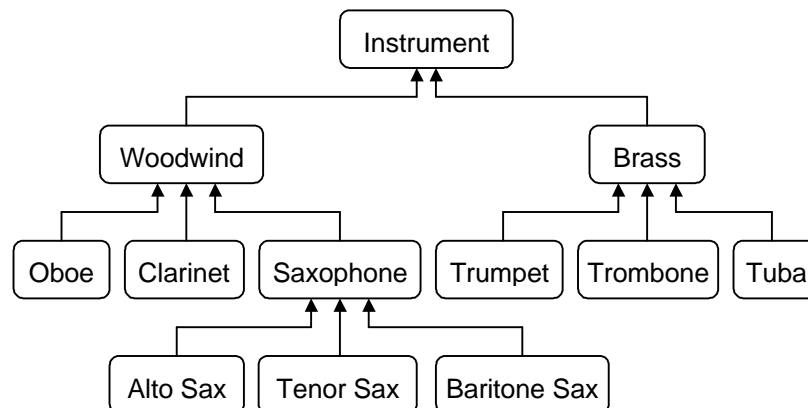


Figure 2. Musical instrument example of inheritance relationship

We could define a series of Java classes for the entries in Figure 2. Each would inherit properties from the superclass entry, while adding its own unique characteristics. In the next section, we'll do this for two entities that have a very simple yet clear inheritance relationship.

Employee and Manager Classes

A manager *is an* employee. A manager has all the general characteristics of an employee (e.g., a name, employee number, date of employment), plus additional characteristics specific to managers (e.g., the number of employees supervised). Let's begin by defining a general-purpose `Employee` class from which a `Manager` subclass can be derived.

Let's implement an `Employee` object with just two data fields, one for the employee's name and another for salary. As well, we'll include six methods. Figure 3 shows the definition of our `Employee` class.

```
1 public class Employee
2 {
3     protected String name;
4     protected double salary;
5
6     public Employee(String s, double d)
7     {
8         name = s;
9         salary = d;
10    }
11
12    // other methods
13
14    public String getName()    { return name;    }
15    public double getSalary() { return salary; }
16    public String toString()  { return name + ", $" + salary; }
17
18    // set a new salary
19    public void setSalary(double newSalary)
20    {
21        salary = newSalary;
22    }
23
24    // determine if 'this' employee is higher paid than 'e'
25    public boolean isHigherPaidThan(Employee e)
26    {
27        return this.salary > e.salary;
28    }
29 }
```

Figure 3. `Employee.java`

There are a few features in the `Employee` class not found in our earlier definition of the `City` class. First, the name and salary data fields in lines 3-4 are defined with the "[protected](#)" [visibility modifier](#), whereas the `City` data fields were defined as `private`. A `private` data field cannot be accessed outside the class, thus providing the ultimate in encapsulation or data hiding. At the opposite end of the spectrum is `public`. Variables or data fields defined as `public` are visible, or accessible, anywhere the class is visible, and they are inherited by all subclasses. This is generally not a good idea for the data fields in an object, for reasons noted earlier. The `protected` modifier is a compromise between `private` and `public`. When a data field is declared `protected`, it is accessible in the class itself, plus it is accessible to, and

inherited by, subclasses of the class.¹ It is not accessible to other classes, however. Thus, declaring these fields `protected` is useful in anticipation of the `Manager` subclass described shortly.

The other new feature is found in the `isHigherPaidThan()` method in lines 25-28. This is a simple method that compares the salaries of two `Employee` objects. It returns a `boolean`: `true` if the first employee's salary is higher than that of the second, `false` otherwise. Let's examine this method by first instantiating two `Employee` objects:

```
Employee newHire = new Employee("Williams", 25000);
Employee oldHat = new Employee("Andrews", 27750);
```

The following code fragment prints the name of the employee with the higher salary:

```
if (newHire.isHigherPaidThan(oldHat))
    System.out.println(newHire.getName() + " makes more");
else
    System.out.println(oldHat.getName() + " makes more");
```

Let's examine the code for the `isHigherPaidThan()` method:

```
public boolean isHigherPaidThan(Employee e)
{
    return this.salary > e.salary;
}
```

The method receives an `Employee` object as an argument. Within the method definition, this object is named `e`. The method contains one line with the following relational expression:

```
this.salary > e.salary
```

This is a simple numeric comparison using the greater than (`>`) relational operator. The values compared are those in the `salary` fields in the two objects. The reserved word `this` is an implicit reference to the current (receiving) object; it is the object variable preceding the dot where the method is called, in this case `newHire`. The expression `this.salary` is the salary field of the `newHire` object. The expression `e.salary` is the salary field of the `Employee` object passed as an argument to the method, in this case `oldHat`.

Before we extend the `Employee` class, let's exercise the class with a test program. The program `EmployeeTest` reads employee information from the standard input into a dynamic array of `Employee` objects named `workers`. The array is then scanned for the highest paid and lowest paid employee, and the result is printed (see Figure 4).

¹ The data field is also accessible to, and inherited by, other code in the same package, but this distinction is not elaborated on here.

```

1  import java.io.*;
2  import java.util.*;
3
4  public class EmployeeTest
5  {
6      public static void main(String[] args) throws IOException
7      {
8          BufferedReader stdin =
9              new BufferedReader(new InputStreamReader(System.in), 1);
10
11         // read workers from standard input, put in dynamic array
12         Vector workers = new Vector();
13         String name;
14         while (!(name = stdin.readLine()).equals("#"))
15         {
16             double salary = Double.parseDouble(stdin.readLine());
17             Employee e = new Employee(name, salary);
18             workers.addElement(e);
19         }
20
21         // find highest and lowest paid workers
22         Employee high = (Employee)workers.elementAt(0);
23         Employee low = high;
24         for (int i = 1; i < workers.size(); i++)
25         {
26             Employee e = (Employee)workers.elementAt(i);
27             if (!high.isHigherPaidThan(e))
28                 high = e;
29             if (low.isHigherPaidThan(e))
30                 low = e;
31         }
32
33         // print results
34         System.out.println(workers.size() + " employees");
35         System.out.println("Highest paid: " + high);
36         System.out.println("Lowest paid: " + low);
37     }
38 }

```

Figure 4. EmployeeTest.java

If a file named test.dat exists with the following employee information

```

Smith
4565.34
Jones
4323.56
Baker
3954.67
Cook
5432.99
King
4321.34
#

```

the EmployeeTest program could be invoked as follows: (User input is underlined.)

```
PROMPT>java EmployeeTest < test.dat
5 employees
Highest paid: Cook, $5432.99
Lowest paid: Baker, $3954.67
```

This program pulls together many pieces in the Java language. Employee information is read into a dynamic array — a `Vector` object — declared in line 12 as `workers`. The work is done in lines 14-19 through a sequence of calls to `readLine()`. We assume the name and salary information are on separate lines: name, then salary, name, then salary, and so on. A line containing `"#"` serves as an end-of-data flag, also called a *sentinel*. A name is read into a `String` object named `name` (line 14) and a salary is read into a `double` variable named `salary` (line 16). Once each name/salary pair is read, an `Employee` object is instantiated (line 17) and then passed to the `addElement()` method of the `Vector` class via the instance variable `workers` (line 18). For clarity, these actions are kept separate, but it is more common to combine them:

```
workers.addElement(new Employee(name, salary));
```

`Employee` objects are added to the `workers` array until `"#"` is returned by the `readLine()` method.

The next part of the program scans through the array to find the highest paid and lowest paid employees (lines 22-31). We begin by retrieving the first `Employee` object from the array and assigning it to `Employee` objects `high` and `low` (lines 22-23). These are replaced by "new" highest and lowest paid employees as the array is scanned. Note that no new `Employee` objects are instantiated in lines 22-23. Both `high` and `low` are object references that "refer to" objects in the `workers` array. Note, as well, that the object reference returned by the `elementAt()` method must be cast (line 22) as noted earlier in our discussion of the `Vector` class.

The array is scanned using a simple `for` loop. `Employee` objects are sequentially retrieved from the array and a reference is assigned to an `Employee` variable `e`. The `isHigherPaidThan()` method is used twice: to determine if the current employee is the new highest paid (line 27) or lowest paid (line 29) employee. If so, the variable `high` or `low` is updated, as appropriate.

Finally, the results are printed in lines 34-36. Note that within the `println()` method the expression `high` is equivalent to `high.toString()`. The `toString()` method is automatically invoked to generate a string representation of the object as part of the string concatenation operation.

The extends Modifier

A class is defined as a subclass of another using an `extends` clause. Since all classes are subclasses of the `Object` class, an `"extends Object"` clause is assumed. So the following class definition

```
class Bicycle extends Object
{
    ...
}
```

is equivalent to

```

class Bicycle
{
    ...
}

```

If a class is to extend a class further down Java's inheritance hierarchy, then an explicit extends clause is required. For this example, we wish to define a `Manager` class that extends the `Employee` class resulting in the inheritance relationship shown in Figure 5.

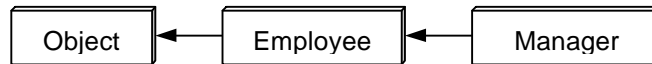


Figure 5. Inheritance relationship for the `Manager` class

This relationship is setup by an extends clause when the `Manager` class is defined:

```

public class Manager extends Employee
{
    ...
}

```

Let's add just one data field to a `Manager` object, a "bonus" representing the percent of the manager's salary issued as a year-end bonus. As well, we'll add methods to retrieve or set the bonus value and to compute the amount in the year-end bonus cheque. The definition of our `Manager` class is shown in Figure 6.

```

1 public class Manager extends Employee
2 {
3     private double bonus;
4
5     public Manager(String id, double pay, double startBonus)
6     {
7         super(id, pay);
8         bonus = startBonus;
9     }
10
11     // other methods
12     public double getBonus() { return bonus; }
13     public void setBonus(double newBonus) { bonus = newBonus; }
14     public String toString()
15     {
16         return name + ", $" + salary + ", " + bonus + "%";
17     }
18
19     public double bonusCheque()
20     {
21         return salary * (bonus / 100.0);
22     }
23 }

```

Figure 6. `Manager.java`

The extends clause appears in the first line of the definition, as expected. The new data field, `bonus`, is declared in line 3 as a private variable of type `double`. As a private data field, the `bonus` field may only be accessed by the methods of the `Manager` class.

The constructor appears in lines 5-9. Here, we see the reserved word `super` as the name of a method. The `super()` method is Java's generic "superclass constructor". Since the `Manager`

class extends the `Employee` class, it is important to ensure the integrity of the inheritance. The `super()` constructor calls the constructor of the super class — in this case, `Employee()` — to instantiate the portion of the `Manager` object that inherits characteristics of an `Employee` object. The arguments passed are the name (`id`) and salary (`pay`) of the manager (line 7). The additional bonus field for managers is initialized as expected, by assigning the argument passed to the constructor (`startBonus`) to the bonus field of the `Manager` object (line 8).

The `getBonus()` and `setBonus()` methods are similar in form to the `getSalary()` and `setSalary()` methods of the `Employee` class met earlier.

The `toString()` method generates a simple string representation of a `Manager` object. Note the use of the data fields `name` and `salary` in line 16. These fields are not defined anywhere in the definition of the `Manager` class; they are data fields in the `Employee` class. However, since the `Manager` class extends the `Employee` class, and since the `name` and `salary` field were declared "protected" in the definition of the `Employee` class, it is legal to access these fields by name within the `Manager` class.

Finally, the `bonusCheque()` method in lines 19-22 returns a double equal to the amount of the manager's year-end bonus cheque.

The `ManagerTest` program exercises the `Manager` class (see Figure 7).

```
1 public class ManagerTest
2 {
3     public static void main(String[] args)
4     {
5         Manager theBoss = new Manager("Mr. B. Cheese", 55000.00, 6.5);
6
7         System.out.println("Manager: " + theBoss.getName());
8         System.out.println("Salary: $" + theBoss.getSalary());
9         System.out.println("Bonus: " + theBoss.getBonus() + "%");
10        double perk = theBoss.bonusCheque();
11        System.out.println("Year-end bonus cheque: $" + perk);
12        System.out.println();
13
14        System.out.println("Next year...");
15        theBoss.setSalary(56000.00);
16        System.out.println("Salary: $" + theBoss.getSalary());
17
18        theBoss.setBonus(7.0);
19        System.out.println("Bonus: " + theBoss.getBonus() + "%");
20
21        perk = theBoss.bonusCheque();
22        System.out.println("Year-end bonus cheque: $" + perk);
23    }
24 }
```

Figure 7. `ManagerTest.java`

This program generates the following output:

```
Manager: Mr. B. Cheese
Salary: $55000.0
Bonus: 6.5%
Year-end bonus cheque: $3575.0
```

```
Next year...
Salary: $56000.0
Bonus: 7.0%
Year-end bonus cheque: $3920.00
```

Only one `Manager` object is instantiated, `theBoss` in line 5. The rest of the program demonstrates the sort of operations that are valid for `Manager` objects. All the `Manager` class methods are used. As well, the `getName()`, `getSalary()`, and `setSalary()` methods are used. These are methods of the `Employee` class; however, since the `Manager` class extends the `Employee` class, they are also valid methods for a `Manager` object. This characteristic is explored further in the next section.

Polymorphism

An object of an extended class can be used anywhere an object of its superclass can be used. This capability is known as *polymorphism*. This is a fancy way of saying the object can have many (*poly-*) forms (*-morph*). So, a `Manager` object can be used anywhere an `Employee` can be used. (Surely, anything that applies to an employee, applies to a manager!) We saw an example in the `ManagerTest` program where a `Manager` object invoked a method of the `Employee` class:

```
theBoss.setSalary(56000.00);
```

The object variable `theBoss` refers to a `Manager` object, yet the `setSalary()` method is defined in the `Employee` class.

Not only can `Manager` objects use methods of the `Employee` class, `Manager` objects can appear as arguments anywhere an `Employee` argument is expected:

```
Manager vicePres = new Manager("Able", 33000.00, 5.6);
Employee salesRep = new Employee("Baker", 35000.00);
if (salesRep.isHigherPaidThan(vicePres))
    System.out.println("Sales rep makes more");
else
    System.out.println("VP makes more");
```

The `isHigherPaidThan()` method compares the salary of two `Employee` objects to determine which salary is higher. It is defined to receive an `Employee` object as an argument (see Figure 3, line 25). In the code fragment above, the argument is `vicePres`, which is a `Manager` object. Through polymorphism, a `Manager` object may substitute for an `Employee` object.

Polymorphism also reveals itself in object assignment. Since a `Manager` object can be used anywhere an `Employee` object can be used, the following code fragment makes sense:

```
Manager m = new Manager("Able", 33000.00, 5.5);
Employee e = m;
```

After the two lines above execute, both `m` and `e` hold references to the same object. Since `m` is a `Manager` reference, it may be used with any method of the `Manager` class (or of the `Employee` class), for example

```
m.setBonus(5.9);
```

However `e` is an `Employee` reference. So

```
e.setSalary(35000.00); // setSalary() is an Employee class method
```

is legal, but

```
e.setBonus(7.0); // WRONG!
//setBonus() is a Manager class method
```

is not.