Appendix C

Creating Classes from Other Classes

Contents

Prerequisite

Appendix B Java Classes

A major advantage of object-oriented programming is the ability to use existing classes when defining new classes. That is, you use classes that you or someone else has written to create new classes, rather than reinventing everything yourself. We begin this appendix with two ways to accomplish this feat.

In the first way, you simply declare an instance of an existing class as a data field of your new class. In fact, you have done this already if you have ever defined a class that had a string as a data field. Since your class is composed of objects, this technique is called composition.

The second way is to use inheritance, whereby your new class inherits properties and behaviors from an existing class, augmenting or modifying them as desired. This technique is more complicated than composition, so we will devote more time to it. As important as inheritance is in Java, you should not ignore composition as a valid and desirable technique in many situations, because inheritance can be used to violate the integrity of an ADT.

Both composition and inheritance define a relationship between two classes. These relationships are often called, respectively, *has a* and *is a* relationships. You will see why when we discuss them in this appendix.

Composition

C.1 Appendix B introduced you to the class `Name` to represent a person's name. It defines constructors, accessor methods, and mutator methods that involve the person's first and last names. The data fields in `Name` are instances of the class `String`. A class uses **composition** when it has a data field that is an instance of another class. And since the class `Name` has an instance of the class `String` as a data field, the relationship between `Name` and `String` is called a ***has a*** relationship.

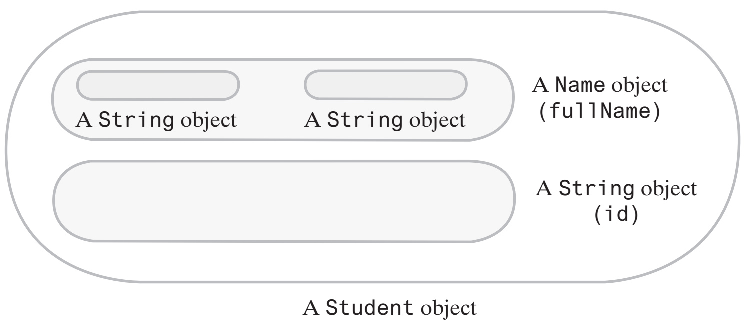
Let's create another class that uses composition. Consider a class of students, each of whom has a name and an identification number. Thus, the class `Student` contains two objects as data fields: an instance of the class `Name` and an instance of the class `String`:

```
private Name   fullName;
private String id;
```

Figure C-1 shows an object of type `Student` and its data fields. Notice that the `Name` object has two `String` objects as its data fields. It is important to realize that these data fields actually contain references to objects, not the objects themselves.

For methods, we give the class `Student` constructors, accessors, mutators, and `toString`. Recall that `toString` is invoked when you use `System.out.println` to display an object, so it is a handy method to include in your class definitions.

FIGURE C-1 A `Student` object is composed of other objects



> **Note: Composition (*has a*)**
> A class uses composition when it has objects as data fields. The class's implementation has no special access to such objects and must behave as a client would. That is, the class must use an object's methods to manipulate the object's data. Since the class "has a," or contains, an instance (object) of another class, the classes are said to have a *has a* relationship.

C.2 Look at the definition of the class `Student` in Listing C-1, and then we will make a few more observations.

LISTING C-1 The class `Student`

```
public class Student
{
    private Name   fullName;
    private String id;      // Identification number

```

```
    public Student()
    {
       fullName = new Name();
       id = "";
    } // end default constructor

    public Student(Name studentName, String studentId)
    {
       fullName = studentName;
       id = studentId;
    }  // end constructor

    public void setStudent(Name studentName, String studentId)
    {
       setName(studentName); // Or fullName = studentName;
       setId(studentId);     // Or id = studentId;
    }  // end setStudent

    public void setName(Name studentName)
    {
       fullName = studentName;
    } // end setName

    public Name getName()
    {
       return fullName;
    } // end getName

    public void setId(String studentId)
    {
       id = studentId;
    } // end setId

    public String getId()
    {
       return id;
    } // end getId

    public String toString()
    {
       return id + " " + fullName.toString();
    } // end toString
}   // end Student
```

The method `setStudent` is useful when we create a student object by using the default constructor or if we want to change both the name and identification number that we gave to a student object earlier. Notice that the method invokes the other set methods from this class to initialize the data fields. For example, to set the field `fullName` to the parameter `studentName`, `setStudent` uses the statement

```
setName(studentName);
```

We could also write this statement as

```
this.setName(studentName);
```

where `this` refers to the instance of `Student` that receives the call to the method `setStudent`. Or we could write the assignment statement

```
fullName = studentName;
```

Implementing methods in terms of other methods is usually desirable.

Suppose that we want `toString` to return a string composed of the student's identification number and name. It must use methods in the class `Name` to access the name as a string. For example, `toString` could return the desired string by using either

```
return id + " " + fullName.getFirst() + " " + fullName.getLast();
```

or, more simply,

```
return id + " " + fullName.toString();
```

The data field `fullName` references a `Name` object whose private fields are not accessible by name in the implementation of the class `Student`. We can access them indirectly via the accessor methods `getFirst` and `getLast` or by invoking `Name`'s `toString` method.

? STUDY

Question 1 What data fields would you use in the definition of a class `Address` to represent a student's address?

Question 2 Add a data field to the class `Student` to represent a student's address. What new methods should you define?

Question 3 What existing methods need to be changed in the class `Student` as a result of the added field that Question 2 described?

Question 4 What is another implementation for the default constructor that uses `this`, as Described in Segment B.25 of Appendix B?

Adapters

C.3 Suppose that you have a class, but the names of its methods do not suit your application. Or maybe you want to simplify some methods or eliminate others. You can use composition to write a new class that has an instance of your existing class as a data field and defines the methods that you want. Such a new class is called an **adapter class**.

For example, suppose that instead of using objects of the class `Name` to name people, we want to use simple nicknames. We could use strings for nicknames, but like `Name`, the class `String` has more methods than we need. The class `NickName` in Listing C-2 has an instance of the class `Name` as a data field, a default constructor, and set and get methods. Arbitrarily, we use the first-name field of the class `Name` to store the nickname.

LISTING C-2 The class `NickName`

```
public class NickName
{
   private Name nick;

   public NickName()
   {
      nick = new Name();
   } // end default constructor

   public void setNickName(String nickName)
   {
      nick.setFirst(nickName);
   } // end setNickName

   public String getNickName()
   {
```

```
        return nick.getFirst();
    } // end getNickName
} // end NickName
```

Notice how this class uses the methods of the class `Name` to implement its methods. A `NickName` object now has only `NickName`'s methods, and not the methods of `Name`.

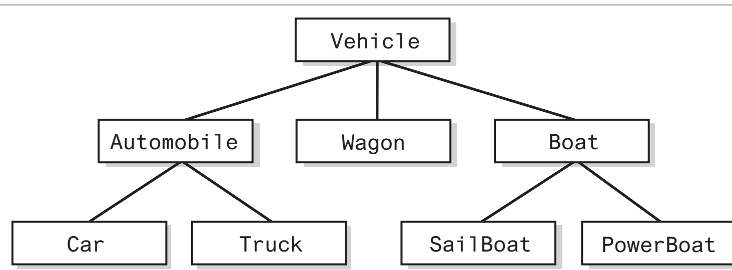
Question 5 Write statements that define `bob` as an instance of `NickName` to represent the nickname *Bob*. Then, using `bob`, write a statement that displays *Bob*.

Inheritance

C.4 **Inheritance** is an aspect of object-oriented programming that enables you to organize classes. The name comes from the notion of inherited traits like eye color, hair color, and so forth, but it is perhaps clearer to think of inheritance as a classification system. Inheritance allows you to define a general class and then later to define more specialized classes that add to or revise the details of the older, more general class definition. This saves work, because the specialized class inherits all the properties of the general class and you need only program the new or revised features.

For example, you might define a class for vehicles and then define more specific classes for particular types of vehicles, such as automobiles, wagons, and boats. Similarly, the class of automobiles includes the classes of cars and trucks. Figure C-2 illustrates this hierarchy of classes. The `Vehicle` class is the **superclass** for the **subclasses**, such as `Automobile`. The `Automobile` class is the superclass for the subclasses `Car` and `Truck`. Another term for superclass is **base class**, and another term for subclass is **derived class**.

FIGURE C-2 A hierarchy of classes



As you move up in the diagram, the classes are more general. A car is an automobile and therefore is also a vehicle. However, a vehicle is not necessarily a car. A sailboat is a boat and is also a vehicle, but a vehicle is not necessarily a sailboat.

C.5 Java and other programming languages use inheritance to organize classes in this hierarchical way. A programmer can then use an existing class to write a new one that has more features. For example, the class of vehicles has certain properties—like miles traveled—that its data fields record. The class also has certain behaviors—like going forward—that its methods define. The classes `Automobile`, `Wagon`, and `Boat` have these properties and behaviors as well. Everything that is true of all `Vehicle` objects, such as the ability to go forward, is described only once and inherited by the classes `Automobile`, `Wagon`, and `Boat`. The subclasses then add to or revise the properties and behaviors that they inherit. Without inheritance, descriptions of behaviors like going forward would have to be repeated for each of the subclasses `Automobile`, `Wagon`, `Boat`, `Car`, `Truck`, and so on.

Note: Inheritance
Inheritance is a way of organizing classes so that common properties and behaviors can be defined only once for all the classes involved. Using inheritance, you can define a general class and then later define more specialized classes that add to or revise the details of the older, more general class definition.

Since the `Automobile` class is derived from the `Vehicle` class, it inherits all the data fields and public methods of that class. The `Automobile` class would have additional fields for such things as the amount of fuel in the fuel tank, and it would also have some added methods. Such fields and methods are not in the `Vehicle` class, because they do not apply to all vehicles. For example, wagons have no fuel tank.

Inheritance gives an instance of a subclass all the behaviors of the superclass. For example, an automobile will be able to do everything that a vehicle can do; after all, an automobile *is a* vehicle. In fact, inheritance is known as an **is a** relationship between classes. Since the subclass and the superclass share properties, you should use inheritance only when it makes sense to think of an instance of the subclass as also being an instance of the superclass.

Note: An *is a* relationship
With inheritance, an instance of a subclass is also an instance of the superclass. Thus, you should use inheritance only when the *is a* relationship between classes is meaningful.

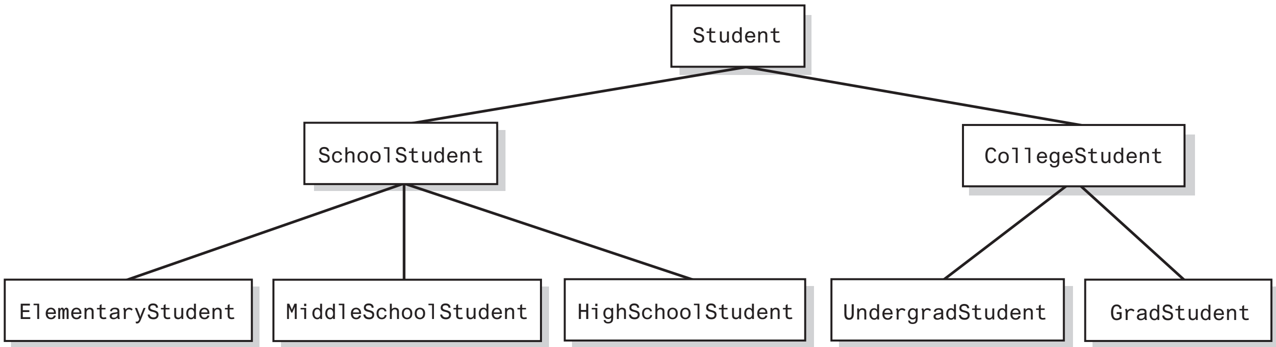
Question 6 Some vehicles have wheels and some do not. Revise Figure C-2 to organize vehicles according to whether they have wheels.

C.6 **Example.** Let's construct an example of inheritance within Java. Suppose we are designing a program that maintains records about students, including those in grade school, high school, and college. We can organize the records for the various kinds of students by using a natural hierarchy that begins with students. College students are then one subclass of students. College students divide into two smaller subclasses: undergraduate students and graduate students. These subclasses might further subdivide into still smaller subclasses. Figure C-3 diagrams this hierarchical arrangement.

A common way to describe subclasses is in terms of family relationships. For example, the class of students is said to be an **ancestor** of the class of undergraduate students. Conversely, the class of undergraduate students is a **descendant** of the class of students.

Although our program may not need any class corresponding to students in general, thinking in terms of such classes can be useful. For example, all students have names, and the methods of initializing, changing, and displaying a name will be the same for all students. In Java, we can define a class that includes data fields for the properties that belong to all subclasses of students. The class likewise will have methods for the behaviors of all students, including methods that manipulate the class's data fields. In fact, we have already defined such a class—`Student`—in Segment C.2.

FIGURE C-3 A hierarchy of student classes



C.7 Now consider a class for college students. A college student is a student, so we use inheritance to derive the class `CollegeStudent` from the class `Student`. Here, `Student` is the existing superclass and `CollegeStudent` is the new subclass. The subclass inherits—and therefore has—all the data fields and methods of the superclass. In addition, the subclass defines whatever data fields and methods we wish to add.

To indicate that `CollegeStudent` is a subclass of `Student`, we write the phrase `extends Student` on the first line of the class definition. Thus, the class definition of `CollegeStudent` begins

```
public class CollegeStudent extends Student
```

When we create a subclass, we define only the added data fields and the added methods. For example, the class `CollegeStudent` has all the data fields and methods of the class `Student`, but we do not mention them in the definition of `CollegeStudent`. In particular, every object of the class `CollegeStudent` has a data field called `fullName`, but we do not declare the data field `fullName` in the definition of the class `CollegeStudent`. The data field is there, however. But because `fullName` is a private data field of the class `Student`, we cannot reference `fullName` directly by name within `CollegeStudent`. We can, however, access and change this data field by using `Student`'s methods, since the class `CollegeStudent` inherits all of the public methods in the superclass `Student`.

For example, if `cs` is an instance of `CollegeStudent`, we can write

```
cs.setName(new Name("Joe", "Java"));
```

even though `setName` is a method of the superclass `Student`. Since we have used inheritance to construct `CollegeStudent` from the class `Student`, every college student *is a* student. That is, a `CollegeStudent` object "knows" how to perform `Student` behaviors.

C.8 A subclass, like `CollegeStudent`, can also add some data fields and/or methods to those it inherits from its superclass. For example, `CollegeStudent` adds the data field `year` and the methods `setYear` and `getYear`. We can set the graduation year of the object `cs` by writing

```
cs.setYear(2019);
```

Suppose that we also add a data field that represents the degree sought and the methods to access and change it. We could also add fields for an address and grades, but to keep it simple, we will not. Let's look at the class as given in Listing C-3 and focus on the constructors first.

LISTING C-3 The class `CollegeStudent`

```
public class CollegeStudent extends Student
{
   private int    year;   // Year of graduation
   private String degree; // Degree sought

   public CollegeStudent()
   {
      super();      // Must be first statement in constructor
      year = 0;
      degree = "";
   } // end default constructor

   public CollegeStudent(Name studentName, String studentId,
                         int graduationYear, String degreeSought)
   {
      super(studentName, studentId); // Must be first
      year = graduationYear;
      degree = degreeSought;
   } // end constructor

   public void setStudent(Name studentName, String studentId,
                          int graduationYear, String degreeSought)
   {
      setName(studentName); // NOT fullName = studentName;
      setId(studentId);     // NOT id = studentId;
// Or setStudent(studentName, studentId); (see Segment C.16)

      year = graduationYear;
      degree = degreeSought;
   } // end setStudent
   <The methods setYear, getYear, setDegree, and getDegree go here.>
   . . .
   public String toString()
   {
      return super.toString() + ", " + degree + ", " + year;
   } // end toString
} // end CollegeStudent
```

Invoking Constructors from Within Constructors

C.9 **Calling the superclass's constructor.** Constructors typically initialize a class's data fields. In a subclass, how can the constructor initialize data fields inherited from the superclass? One way is to call the superclass's constructor. The subclass's constructor can use the reserved word `super` as a name for the constructor of the superclass.

Notice that the default constructor in the class `CollegeStudent` begins with the statement

```
super();
```

This statement invokes the default constructor of the superclass. Our new default constructor must invoke the superclass's default constructor to properly initialize the data fields that are inherited from the superclass. Actually, if you do not invoke `super`, Java will do it for you. In this book, we will always invoke `super` explicitly, to make the action a bit clearer. Note that the call to `super` must occur first in the constructor. You can use `super` to invoke a constructor only from within another constructor.

In like fashion, the second constructor invokes a corresponding constructor in the superclass by executing the statement

```
super(studentName, studentId);
```

If you omit this statement, Java will invoke the default constructor, which is *not* what you want.

> **Note: Calling the constructor of the superclass**
> You can use `super` within the definition of a constructor of a subclass to call a constructor of the superclass explicitly. When you do, `super` always must be the first action taken in the constructor definition. You cannot use the name of the constructor instead of `super`. If you omit `super`, each constructor of a subclass automatically calls the default constructor of the superclass. Sometimes this action is what you want, but sometimes it is not.

> **Note: Constructors are not inherited**
> A constructor of a class `C` creates an object whose type is `C`. It wouldn't make sense for this class to have a constructor named anything other than `C`. But that is what would happen if a class like `CollegeStudent` inherited `Student`'s constructors: `CollegeStudent` would have a constructor named `Student`.
>
> Even though `CollegeStudent` does not inherit `Student`'s constructors, its constructors do call `Student`'s constructors, as you have just seen.

C.10 **Reprise: Using `this` to invoke a constructor.** As you saw in Segment B.25 of Appendix B, you use the reserved word `this` much as we used `super` here, except that it calls a constructor of the same class instead of a constructor of the superclass. For example, consider the following definition of a constructor that we might add to the class `CollegeStudent` in Segment C.8:

```
public CollegeStudent(Name studentName, String studentId)
{
   this(studentName, studentId, 0, "");
} // end constructor
```

The one statement in the body of this constructor definition is a call to the constructor whose definition begins

```
public CollegeStudent(Name studentName, String studentId,
                      int graduationYear, String degreeSought)
```

As with `super`, any use of `this` must be the first action in a constructor definition. Thus, a constructor definition cannot both a call using `super` and a call using `this`. What if you want both a call with `super` and a call with `this`? In that case, you would use `this` to call a constructor that has `super` as its first action.

Private Fields and Methods of the Superclass

C.11 **Accessing inherited data fields.** The class `CollegeStudent` has a `setStudent` method with four parameters, `studentName`, `studentId`, `graduationYear`, and `degreeSought`. To initialize the inherited data fields `fullName` and `id`, the method invokes the inherited methods `setName` and `setId`:

```
setName(studentName); // NOT fullName = studentName
setId(studentId);     // NOT id = studentId
```

Recall that `fullName` and `id` are private data fields defined in the superclass `Student`. Only a method in the class `Student` can access `fullName` and `id` directly by name from within its definition. Although the class `CollegeStudent` inherits these data fields, none of its methods can access them by name. Thus, `setStudent` cannot use an assignment statement such as

```
id = studentId; // ILLEGAL in CollegeStudent's setStudent
```

to initialize the data field `id`. Instead it must use some public mutator method such as `setId`.

Note: A data field that is private in a superclass is not accessible by name within the definition of a method for any other class, including a subclass. Even so, a subclass inherits the data fields of its superclass.

The fact that you cannot access a private data field of a superclass from within the definition of a method of a subclass seems wrong to people. To do otherwise, however, would make the access modifier `private` pointless: Anytime you wanted to access a private data field, you could simply create a subclass and access it in a method of that class. Thus, all private data fields would be accessible to anybody who was willing to put in a little extra effort.

C.12 **Private methods of the superclass.** A subclass cannot invoke a superclass's private methods directly. This should not be a problem, since you should use private methods only as helpers within the class in which they are defined. That is, a class's private methods do not define behaviors. Thus, we say that a subclass does not inherit the private methods of its superclass. If you want to use a superclass's method in a subclass, you should make the method either protected or public. We discuss protected methods in Java Interlude 7.

Suppose that superclass `B` has a public method `m` that calls a private method `p`. A class `D` derived from `B` inherits the public method `m`, but not `p`. Even so, when a client of `D` invokes `m`, `m` calls `p`. Thus, a private method in a superclass still exists and is available for use, but a subclass cannot call it directly by name.

Note: A subclass does not inherit and cannot invoke by name a private method of the superclass.

Overriding and Overloading Methods

C.13 The set and get methods of the class `CollegeStudent` are straightforward, so we will not bother to look at them. However, we have provided the class with a method `toString`. Why did we do this, when our new class inherits a `toString` method from its superclass `Student`? Clearly, the string that the superclass's `toString` method returns can include the student's name and identification number, but it cannot include the year and degree that are associated with the subclass. Thus, we need to write a new method `toString`.

But why not have the new method invoke the inherited method? We can do this, but we'll need to distinguish between the method that we are defining for `CollegeStudent` and the method inherited from `Student`. As you can see from the class definition in Segment C.8, the new method `toString` contains the statement

```
return super.toString() + ", " + degree + ", " + year;
```

Since `Student` is the superclass, we write

```
super.toString()
```

to indicate that we are invoking the superclass's `toString`. If we omitted `super`, our new version of `toString` would invoke itself. Here we are using `super` as if it were an object. In contrast, we used `super` with parentheses as if it were a method within the constructor definitions.

If you glance back at Segment C.2, you will see that `Student`'s `toString` method appears as follows:

```
public String toString()
{
   return id + " " + fullName.toString();
} // end toString
```

This method calls the `toString` method defined in the class `Name`, since the object `fullName` is an instance of the class `Name`.

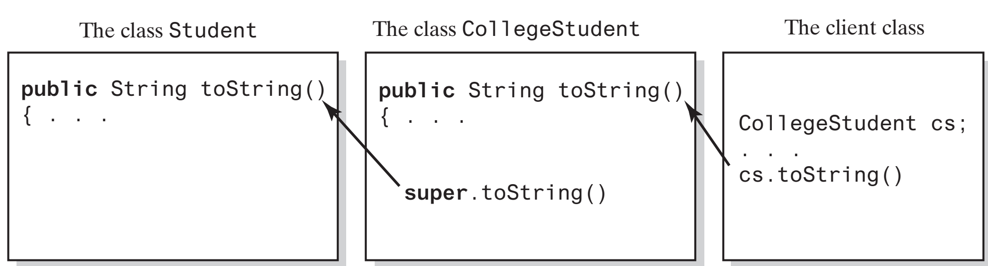
C.14 **Overriding a method.** In the previous segment, you saw that the class `CollegeStudent` defines a method `toString` and also inherits a method `toString` from its superclass `Student`. Both of these methods have no parameters. The class, then, has two methods with the same name, the same parameters, and the same return type.

When a subclass defines a method with the same name, the same number and types of parameters, and the same return type as a method in the superclass, the definition in the subclass is said to **override** the definition in the superclass. Objects of the subclass that invoke the method will use the definition in the subclass. For example, if `cs` is an instance of the class `CollegeStudent`,

```
cs.toString()
```

uses the definition of the method `toString` in the class `CollegeStudent`, not the definition of `toString` in the class `Student`, as Figure C-4 illustrates. As you've already seen, however, the definition of `toString` in the subclass can invoke the definition of `toString` in the superclass by using `super`.

FIGURE C-4 The method `toString` in `CollegeStudent` overrides the method `toString` in `Student`



Note: Overriding a method definition
A method in a subclass overrides a method in the superclass when both methods have the same name, the same number and types of parameters, and the same return type. Since a method's signature is its name and parameters, a method in a subclass overrides a method in the superclass when both methods have the same signature and return type.

Note: Overriding and access
An overriding method in a subclass can have either public, protected, or package access according to the access of the overridden method in the superclass, as follows:

Access of the overridden method in the superclass	Access of the overriding method in the subclass
public	public
protected	protected or public
package	package, protected, or public

A private method in a superclass cannot be overridden by a method in a subclass.

Note that Segment J7.2 in Java Interlude 7 discusses protected access, and Segment B.34 of Appendix B describes package access.

Note: You can use `super` in a subclass to call an overridden method of the superclass.

Question 7 If a subclass overrides a protected method in its superclass, can the overriding method be

a. Public?
b. Protected?
c. Private?

Question 8 If a subclass overrides a public method in its superclass, can the overriding method be

a. Public?
b. Protected?
c. Private?

Question 9 Question 5 asked you to create an instance of `NickName` to represent the nickname *Bob*. If that object is named `bob`, do the following statements produce the same output? Explain.

```
System.out.println(bob.getNickName());
System.out.println(bob);
```

C.15 **Covariant return types (optional).** A class cannot define two methods that have different return types but the same signatures—that is, the same name and parameters. However, if the two methods are in different classes, and one class is a subclass of the other, this can be possible. In particular, when a method in a subclass overrides a method in the superclass, their signatures are the same. But the return type of the method in the subclass can be a subclass of the return type of the method in the superclass. Such return types are said to be **covariant**.

For example, in Segment C.8 the class `CollegeStudent` was derived from the class `Student` defined in Segment C.2. Now imagine a class `School` that maintains a collection of `Student` objects. (This book will give you the tools to actually do this.) The class has a method `getStudent` that returns a student given his or her ID number. The class might appear as follows:

```
public class School
{
   . . .
   public Student getStudent(String studentId)
   {
      . . .
   } // end getStudent
} // end School
```

Now, consider a class `College` that has a collection of college students. We can derive `College` from `School` and override the method `getStudent`, as follows:

```
public class College extends School
{
   . . .
   public CollegeStudent getStudent(String studentId)
   {
      . . .
   } // end getStudent
} // end College
```

The method `getStudent` has the same signature as `getStudent` in `School`, but the return types of the two methods differ. In fact, the return types are covariant—and therefore legal—because `CollegeStudent` is a subclass of `Student`.

C.16 **Reprise: Overloading a method.** Segment B.29 of Appendix B discussed overloaded methods within the same class. Such methods have the same name but different signatures. Java is able to distinguish between these methods since their parameters are not identical.

Suppose that a subclass has a method with the same name as a method in its superclass, but the methods' parameters differ in number or data type. The subclass would have both methods—the one it defines and the one it inherits from the superclass. The method in the subclass overloads the method in the superclass.

For example, the superclass `Student` and the subclass `CollegeStudent` each have a method named `setStudent`. The methods are not exactly the same, however, as they have a different number of parameters. In `Student`, the method's header is

```
public void setStudent(Name studentName, String studentId)
```

whereas in `CollegeStudent` it is

```
public void setStudent(Name studentName, String studentId,
                       int graduationYear, String degreeSought)
```

An instance of the class `Student` can invoke only `Student`'s version of the method, but an instance of `CollegeStudent` can invoke either method. Again, Java can distinguish between the two methods because they have different parameters.

Within the class `CollegeStudent`, the implementation of `setStudent` can invoke `Student`'s `setStudent` to initialize the fields `fullName` and `id` by including the statement

```
setStudent(studentName, studentId);
```

instead of making calls to the methods `setName` and `setId`, as we did in Segment C.8. Since the two versions of `setStudent` have different parameter lists, we do not need to preface the call with `super` to distinguish the two methods. However, we are free to do so by writing

```
super.setStudent(studentName, studentId);
```

> **Note: Overloading a method definition**
> A method in a class overloads another method in either the same class or its superclass when both methods have the same name but differ in the number or types of parameters. Thus, overloaded methods have the same name but different signatures.

Although the terms "overloading" and "overriding" are easy to confuse, you should distinguish between the concepts, as they both are important.

C.17 **Multiple use of `super`.** As we have already noted, within the definition of a method of a subclass, you can call an overridden method of the superclass by prefacing the method name with `super` and a dot. However, if the superclass is itself derived from some other superclass, you cannot repeat the use of `super` to invoke a method from that superclass.

For example, suppose that the class `UndergradStudent` is derived from the class `CollegeStudent`, which is derived from the class `Student`. You might think that you can invoke a method of the class `Student` within the definition of the class `Undergraduate`, by using `super.super`, as in

```
super.super.toString(); // ILLEGAL!
```

As the comment indicates, this repeated use of `super` is not allowed in Java.

Note: `super`
Although a method in a subclass can invoke an overridden method defined in the superclass by using `super`, the method cannot invoke an overridden method that is defined in the superclass's superclass. That is, the construct `super.super` is illegal.

Question 10 Are the two definitions of the constructors for the class `Student` (Segment C.2) an example of overloading or overriding? Why?

Question 11 If you add the method

```
public void setStudent(Name studentName, String studentId)
```

to the class `CollegeStudent` and let it give some default values to the fields `year` and `degree`, are you overloading or overriding `setStudent` in the class `Student`? Why?

C.18 **The `final` modifier.** Suppose that a constructor calls a public method `m`. For simplicity, imagine that this method is in the same class `C` as the constructor, as follows:

```
public class C
{
   . . .
   public C()
   {
      m();
      . . .
   } // end default constructor
   public void m()
   {
      . . .
   } // end m
   . . .
```

Now imagine that we derive a new class from `C` and we override the method `m`. If we invoke the constructor of our new class, it will call the superclass's constructor, which will call our overridden version of the method `m`. This method might use data fields that the constructor has not yet initialized, causing an error. Even if no error occurs, we will, in effect, have altered the behavior of the superclass's constructor.

To specify that a method definition cannot be overridden with a new definition in a subclass, you make it a **final method** by adding the `final` modifier to the method header. For example, you can write

```
public final void m()
```

Note that private methods are automatically final methods, since you cannot override them in a subclass.

Programming Tip: If a constructor invokes a public method in its class, declare that method to be final so that no subclass can override the method and hence change the behavior of the constructor.

Constructors cannot be final. Since a subclass does not inherit, and therefore cannot override, a constructor in the base case, final constructors are unnecessary.

You can declare an entire class as a **final class**, in which case you cannot use it as superclass to derive any other class from it. Java's `String` class is an example of a final class.

Note: `String` cannot be the superclass for any other class because it is a final class.

Programming Tip: When you design a class, consider the classes derived from it, either now or in the future. They might need access to your class's data fields. If your class does not have public accessor or mutator methods, provide protected versions of such methods. Keep the data fields private. Protected access is discussed in Java Interlude 7.

Multiple Inheritance

C.19 Some programming languages allow one class to be derived from two different superclasses. That is, you can derive class `C` from classes `A` and `B`. This feature, known as **multiple inheritance**, is not allowed in Java. In Java, a subclass can have only one superclass. You can, however, derive class `B` from class `A` and then derive class `C` from class `B`, since this is not multiple inheritance.

A subclass can implement any number of interfaces—which we describe in the prelude to this book—in addition to extending any one superclass. This capability gives Java an approximation to multiple inheritance without the complications that arise with multiple superclasses.

Type Compatibility and Superclasses

C.20 **Object types of a subclass.** Previously, you saw the class `CollegeStudent`, which was derived from the class `Student`. In the real world, every college student is also a student. This relationship holds in Java as well. Every object of the class `CollegeStudent` is also an object of the class `Student`. Thus, if we have a method that has a parameter of type `Student`, the argument in an invocation of this method can be an object of type `CollegeStudent`.

Specifically, suppose that the method in question is in some class and begins as follows:

```
public void someMethod(Student scholar)
```

Within the body of `someMethod`, the object `scholar` can invoke public methods that are defined in the class `Student`. For example, the definition of `someMethod` could contain the expression `scholar.getId()`. That is, `scholar` has `Student` behaviors.

Now consider an object `joe` of `CollegeStudent`. Since the class `CollegeStudent` inherits all the public methods of the class `Student`, `joe` can invoke those inherited methods. That is, `joe` can behave like an object of `Student`. (It happens that `joe` can do more, since it is an object of `CollegeStudent`, but that is not relevant right now.) Therefore, `joe` can be the argument of `someMethod`. That is, for some object `o`, we can write

```
o.someMethod(joe);
```

No automatic type casting[1] has occurred here. As an object of the class `CollegeStudent`, `joe` is also of type `Student`. The object `joe` need not be, and is not, type-cast to an object of the class `Student`.

We can take this idea further. Suppose that we derive the class `UndergradStudent` from the class `CollegeStudent`. In the real world, every undergraduate is a college student, and every college student is also a student. Once again, this relationship holds for our Java classes. Every object of the class `UndergradStudent` is also an object of the class `CollegeStudent` and so is also an object of the class `Student`. Thus, if we have a method whose parameter is of type `Student`, the argument in an invocation of this method can be an object of type `UndergradStudent`. Thus, an object can actually have several types as a result of inheritance.

Note: An object of a subclass has more than one data type. Everything that works for objects of an ancestor class also works for objects of any descendant class.

C.21 Because an object of a subclass also has the types of all of its ancestor classes, you can assign an object of a class to a variable of any ancestor type, but not the other way around. For example, since the class `UndergradStudent` is derived from the class `CollegeStudent`, which is derived from the class `Student`, the following are legal:

```
Student amy = new CollegeStudent();
Student brad = new UndergradStudent();
CollegeStudent jess = new UndergradStudent();
```

However, the following statements are all illegal:

```
CollegeStudent cs = new Student();           // ILLEGAL!
UndergradStudent ug = new Student();         // ILLEGAL!
UndergradStudent ug2 = new CollegeStudent(); // ILLEGAL!
```

This makes perfectly good sense. For example, a college student is a student, but a student is not necessarily a college student. Some programmers find the phrase “is a” to be useful in deciding what types an object can have and what assignments to variables are legal.

Programming Tip: Because an object of a subclass is also an object of the superclass, do not use inheritance when an *is a* relationship does not exist between your proposed class and an existing class. Even if you want class `C` to have some of the methods of class `B`, use composition if these classes do not have an *is a* relationship.

Question 12 If `HighSchoolStudent` is a subclass of `Student`, can you assign an object of `HighSchoolStudent` to a variable of type `Student`? Why or why not?

Question 13 Can you assign an object of `Student` to a variable of type `HighSchoolStudent`? Why or why not?

The Class `Object`

C.22 As you have already seen, if you have a class `A` and you derive class `B` from it, and then you derive class `C` from `B`, an object of class `C` is of type `C`, type `B`, and type `A`. This works for any chain of subclasses no matter how long the chain is.

Java has a class—named `Object`—that is at the beginning of every chain of subclasses. This class is an ancestor of every other class, even those that you define yourself. Every object of

1. Segment S1.21 of Supplement 1 (online) reviews type casts.

every class is of type `Object`, as well as being of the type of its class and also of the types of all the other ancestor classes. If you do not derive your class from some other class, Java acts as if you had derived it from the class `Object`.

Note: Every class is a descendant class of the class `Object`.

The class `Object` contains certain methods, among which are `toString`, `equals`, and `clone`. Every class inherits these methods, either from `Object` directly or from some other ancestor class that ultimately inherited the methods from the class `Object`.

The inherited methods `toString`, `equals`, and `clone`, however, will almost never work correctly in the classes you define. Typically, you need to override the inherited method definitions with new, more appropriate definitions. Thus, whenever you define the method `toString` in a class, for example, you are actually overriding `Object`'s method `toString`.

C.23 **The `toString` method.** The method `toString` takes no arguments and is supposed to return all the data in an object as a `String`. However, you will not automatically get a nice string representation of the data. The inherited version of `toString` returns a value based upon the invoking object's memory address. You need to override the definition of `toString` to cause it to produce an appropriate string for the data in the class being defined. You might want to look again at the `toString` methods in Segments C.2 and C.8.

C.24 **The `equals` method.** Consider the following objects of the class `Name` that we defined in Appendix B:

```
Name joyce1 = new Name("Joyce", "Jones");
Name joyce2 = new Name("Joyce", "Jones");
Name derek = new Name("Derek", "Dodd");
```

Now `joyce1` and `joyce2` are two distinct objects that contain the same name. Typically, we would consider these objects to be equal, but in fact `joyce1.equals(joyce2)` is false. Since `Name` does not define its own `equals` method, it uses the one it inherits from `Object`. `Object`'s `equals` method compares the addresses of the objects `joyce1` and `joyce2`. Because we have two distinct objects, these addresses are not equal. However, `joyce1.equals(joyce1)` is true, since we are comparing an object with itself. This comparison is an **identity**. Notice that identity and equality are different concepts.

The method `equals` has the following definition in the class `Object`:

```
public boolean equals(Object other)
{
   return (this == other);
} // end equals
```

Thus, the expression `x.equals(y)` is `true` if `x` and `y` reference the same object. We must override `equals` in the class `Name` if we want it to behave more appropriately.

As you will recall, `Name` has two data fields, `first` and `last`, that are instances of `String`. We could decide that two `Name` objects are equal if they have equal first names and equal last names. The following method, when added to the class `Name`, detects whether two `Name` objects are equal by comparing their data fields:

```
public boolean equals(Object other)
{
   boolean result = false;

   if (other instanceof Name)
   {
      Name otherName = (Name)other;
```

```
         result = first.equals(otherName.first) &&
                  last.equals(otherName.last);
      } // end if

      return result;
   } // end equals
```

To ensure that the argument passed to the method `equals` is a `Name` object, you use the Java operator `instanceof`. For example, the expression

```
other instanceof Name
```

is true if `other` references an object of either the class `Name` or a class derived from `Name`. If `other` references an object of any other class, or if `other` is `null`, the expression will be false.

Given an appropriate argument, the method `equals` compares the data fields of the two objects. Notice that we first must cast the type of the parameter `other` from `Object` to `Name` so that we can access `Name`'s data fields. To compare two strings, we use `String`'s `equals` method. The class `String` defines its own version of `equals` that overrides the `equals` method inherited from `Object`.

STUDY

Question 14 If `sue` and `susan` are two instances of the class `Name`, what `if` statement can decide whether they represent the same name?

C.25 **The `clone` method.** Another method inherited from the class `Object` is the method `clone`. This method takes no arguments and returns a copy of the receiving object. The returned object is supposed to have data identical to that of the receiving object, but it is a different object (an identical twin or a "clone"). As with other methods inherited from the class `Object`, we need to override the method `clone` before it can behave properly in our class. However, in the case of the method `clone`, there are other things we must do as well. A discussion of the method `clone` appears in Java Interlude 9.