Chapter 3: Object-Oriented Programming

It is easy for us to manipulate real-world objects such as a clock or, after a certain learning period, a car or a computer. Such objects often contain certain information and provide us with means to manipulate that information. A clock, for example, stores the current time and provides us with controls to set the time and retrieve it by glancing at the clock's display. Moreover, small objects are often part of other objects. Most cars, for example, contain a clock that looks different than a wristwatch but provides the same functionality. Thus, car manufacturers do not have to recreate a clock for every car they produce. Instead, they purchase existing clocks and adjust them for their particular purposes. They do not really need to know anything about clocks, other than how to interface the clock's control mechanisms with the particular design features they would like to implement in their car. And most of us do not know many details about internal combustion engines, transmissions, and hydraulic suspension systems, yet many of us are perfectly comfortable driving a car by manipulating the appropriate controls.

This concept of hiding an object's complexities from a user has worked well in the real world, and software designers want to emulate it. They want to create pieces of software that can contain information and provide mechanisms to manipulate that data without bothering a user with the inner complexities of the code. In creating a large program one can draw on smaller software objects, adjust their interface as needed, and produce a working, error-free application without having to study the inner workings of the original objects.

This approach has appropriately been termed *object-oriented programming*. The goal is to create abstract software objects that exist inside a computer's memory but can be used and manipulated much like real-world objects. Object-oriented programming revolves around six concepts:

- Classes: The fundamental structure of every Java program, containing data fields and mechanisms to manipulate that data. Classes provide blueprints to construct software objects.
- Instances: Objects constructed according to class blueprints that exist in the memory of the computer. One class can be used to instantiate any number of objects.
- Encapsulation: Allowing or disallowing access to the data and mechanisms that a class or object contains.
- Overloading: Providing multiple definitions for certain mechanisms, allowing them to adjust automatically to different situations.
- Inheritance: A mechanism to impose relationships between classes and their corresponding objects.
- Polymorphism: The ability to deal with multiple related classes based on a common feature, giving an entire class hierarchy the ability to adjust to an appropriate situation.

This chapter introduces the basic principles of object-oriented programming as they apply to Java, using concrete examples to highlight the fundamental features of this paradigm. Object-oriented code *is central* to programming in Java, and the concepts introduced in this chapter will form the basis for understanding *every* Java program. Section 3.1 introduces the fundamental concepts of class, object, and instantiation. Section 3.2 describes how to regulate access to the pieces of an object. Section 3.3 describes the process of overloading, while section 3.4 deals with inheritance and creating class hierarchies. Section 3.5 introduces the basic Java Object, the ancestor of every Java class. Section 3.6 covers the remaining object-oriented concepts of polymorphism and interfaces. The last section, which is optional, uses object-oriented design principles to create an invoicing program.

Quick View

Here is a quick overview of the topics covered in this chapter.

3.1. Classes and Objects
Classes; Objects, Instantiation, and Constructors; Destructors and Garbage Collection

3.2. Class Access and Encapsulation
Accessing Fields and Methods; Private, Public, and Protected; Static Fields and Methods

3.3. Overloading
Overloading Methods; Overloading Constructors

3.4. Inheritance
Creating Class Hierarchies; Inheritance and Constructors; Overriding; Abstract and Final

3.5. The Basic Java Object

(*) 3.6 Interfaces and Polymorphism
Multiple Inheritance and Interfaces; Polymorphism

(**) Case Study: OO Design for Invoice Program

(*) This section is optional but recommended
(**) This section is optional

3.1. Classes and Objects

Any code written in Java is entirely composed of classes and objects. We have previously written Java programs and hence we have already created classes.[1] This section defines classes formally and shows how to use them to manufacture objects.

Classes

Classes

A class is the fundamental structure in Java. It is composed of two sections, fields to contain data and methods to manipulate data or perform an action. Every class represents a new reference type that can be used by other classes. Classes are defined using the syntax:

```
[public] class ClassName [extends ClassName] [implements InterfaceList]
{  /* list of fields */
   /* list of methods */;
}
```

[1] `String` and `StringBuffer` in section 1.4, the wrapper classes in section 2.5, the `Console` class in section 2.4, and decimal formatting in section 1.5 are examples where we have used existing non-executable classes.

where `ClassName` *is the name of the class,* `extends` *indicates that the class is derived from another class and* `implements` *indicates that the class has attributes in common with one or more interfaces.*

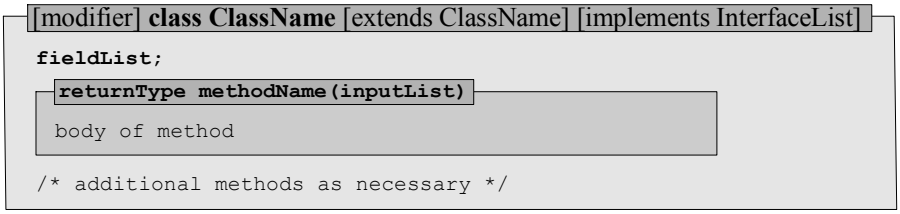


Figure 3.01: Representation of a Class

Every Java program consists of one or more classes that manipulate each other's fields and use their respective methods to accomplish their tasks. Classes containing the method `public static void main(String args[])` are executable. Classes do not need to be executable, and fields and methods in a class do not need to be prefaced by the keywords `public static`.

When creating classes, we should initially be less concerned with the inner workings of the class. We should instead think about all possible uses that our class should allow and worry about *what* we want our class to accomplish not *how* to accomplish it.

Software Engineering Tip: The hardest part in designing a class is not how to implement it, but to determine the fields and methods it should have. The "has-a" and "does-a" questions can help:

- If a class "has-a" property, then the class needs a field to store that information. Use field names that represent the property they are storing.
- If a class "does-a" certain action, then the class needs a method to perform that action. Use method names representative for the actions they perform.

Example 3.01: Simple `Address` *class*

Suppose we are working on an address book program. Design an `Address` class that can store a first name, last name, and email address, and display an address.

Problem Analysis: An address "has-a" first name, last name, and email address, so it needs three fields: one for the first name, one for the last name, and one for the email address. It should perform the action ("does-a") of displaying its value, so it needs at least one method.

Class Implementation: The fields should store strings so they are of type `String`. The method to display an address requires no input because the address information is part of the class as fields and delivers no output because it will display the address on the screen.[2] We name it `showAddress`. The class is not a complete program so we do not need the standard `main` method that was part of all classes in chapter 2.

[2] See section 2.1 for details on defining methods.

```
class Address
{  // Fields
   String firstName;
   String lastName;
   String email;
   // Methods
   void showAddress()
   {  /* implementation */ }
}
```

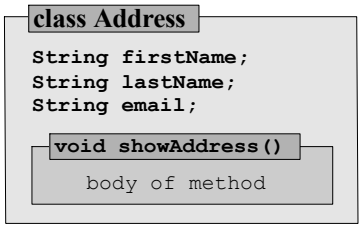


Figure 3.02: `Address` *class*

The class can be saved in a file with any name but it must have a `.java` extension. The preferred file name is `Address.java`.

> **Software Engineering Tip:** To easily recognize classes, you should capitalize the class name and save each class in a separate file with the same name (including capitalization) as the class name, followed by the `.java` extension.[3] Fields should be listed first before methods in a class, using names that start with lowercase letters.

We therefore save our class in the file `Address.java` and compile it by typing `javac Address.java`. It compiles without problems even though the `showAddress` method does not have a method body. ■

Classes can contain fields of the same or of different type.

Example 3.02: Simple `Length` *class*

Suppose we are working on a project involving "length". A length is an object comprised of two data pieces: a numeric value and a scale. Several different scales can be used, such as inches and feet, or the more useful metric scales centimeter and meter. Design a `Length` class that allows conversions from, say, meter to feet and back.

Problem Analysis: Using the above "has-a" and "does-a" phrases we could say:

- a `Length` "has a" value and a scale
- a `Length` "does a" conversion from one scale to another

Therefore, our `Length` class needs at least two data fields and two methods.

Class Implementation: One of the data fields contains the numeric value of the length so we use the type `double`. The second representing the scale is of type `String`. We also need at least two methods: one to convert the current length to meter, the other to convert to feet. Both methods return a `double` value. They do not need any input parameters since they should convert the length stored in the fields of the class. Our first approach to designing a `Length` class is:

[3] A class without modifiers could be saved using an arbitrary file name with the `.java` extension. A class preceded by the `public` modifier *must* be saved in a file named after the class name with the `.java` extension.

```
class Length
{  // Fields
   double value = 0;
   String scale = "none";
   // Methods
   double convertToMeter()
   {  /* implementation */ }
   double convertToFeet()
   { /* implementation */ }
}
```

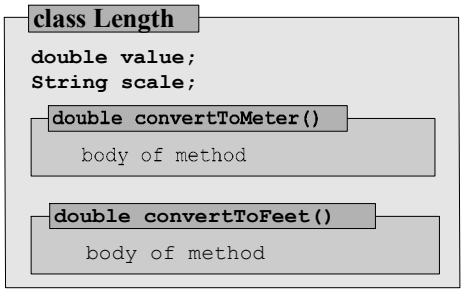


Figure 3.03: `Length` *class*

The methods `convertToMeter` and `convertToFeet` return a `double` value, but their method bodies do not include the `return` keyword. Therefore the class will not compile. It only specifies *what* our class can do without worrying about *how* to do it.

Software Engineering Tip: It is usually more difficult to decide which methods should be part of a class and what action they should perform than it is to determine the fields. Use the following guide when designing methods:

- (1) Can the methods accomplish what the class needs?
- (2) Are the methods flexible enough to work in most possible situations?
- (3) Are the methods easy to use in most possible situations?
- (4) Are the methods restricted to one particular aspect or subtask of our class?
- (5) Are the input and output parameters clearly defined and meaningful?
- (6) Do the class and method names adequately reflect their function?

Looking at items (2) and (5) of this guide we should reconsider the design of our `Length` class:

- We need a mechanism to display a `Length` so that we add a `showLength` method
- The methods `convertToMeter` and `convertToFeet` return a `double` but should really return another `Length`.[4]

```
class Length
{  // fields
   double value = 0;
   String scale = "none";
   // methods
   Length convertToMeter()
   {  /* implementation */ }
   Length convertToFeet()
   { /* implementation */ }
   void showLength()
   { /* implementation */ }
}
```

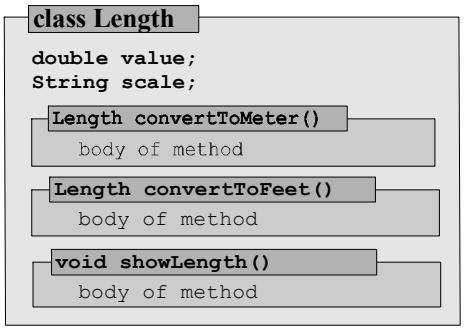


Figure 3.04: New `Length` *class*

■

[4] It might seem impossible for a method in a class `Length` to return a `Length` because the very concept of `Length` is defined in this class for the first time. But `Length` itself is a class while methods return an *object of type* `Length`.

Objects, Instantiation, and Constructors

Classes describe the data to store and the methods to manipulate data but they are only an abstraction of the actual existing entity: the object.

Object

> *An object is an entity that has been manufactured according to the blueprint that a class provides. Objects have the type of the class whose specifications were used to manufacture it. Objects should be seen as real (inside the computer) whereas classes are conceptual.*[5]

The relationship between objects and classes is similar to the relationship between the blueprints of a car (the class) and the actual car (the object) manufactured on the assembly line according to the specifications of the blueprints.

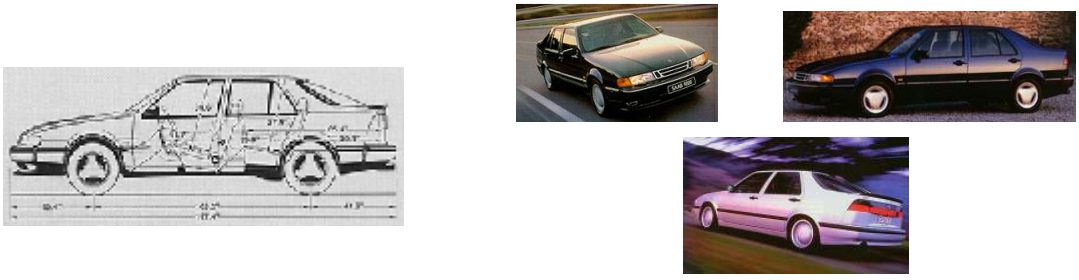


Figure 3.05: Blueprint of a car (Class) versus actual cars (Objects)

One class can be used to manufacture many objects of its type through a process called instantiation. These objects have identical capabilities but different reference names.

Instantiation

> *Instantiation is the process of creating an object according to the blueprint of a class. Memory is allocated to provide storage space for the fields and methods and any necessary initialization is performed. The keyword* `new` *is used to instantiate an object, using the syntax:*
>
> ```
> [ClassName] referenceName = new ClassName(paramList);
> ```
>
> *where* `ClassName` *is the name of the class providing the blueprint for the object and* `referenceName` *is the name of the object to be instantiated. Each new object gets its own set of data field.*[6] *Methods are created once and shared by objects instantiated from the same class.*

[5] In many situations the distinction between class and object is not necessary. We will occasionally use the word object when we really should use class.
[6] Fields can be prefaced by the `static` modifier, in which case they are shared between objects of the same type.

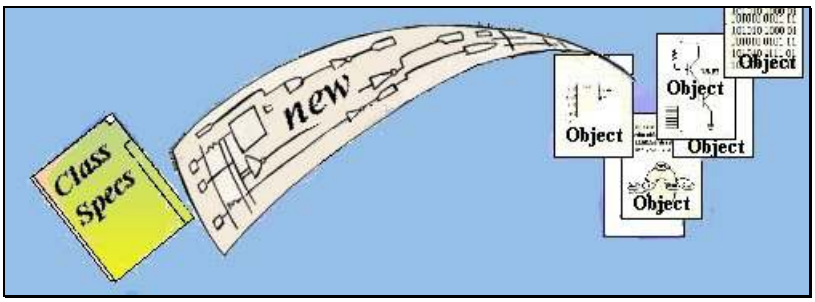


Figure 3.06: Instantiation process

Example 3.03: Instantiating `Address` *objects*

Create a class containing the standard `main` method used in chapter 2. Instantiate three objects of type `Address` inside that method, where the class `Address` is as defined in example 3.01. Compile and execute the new class.

We have already defined the `Address` class so we need to create another class called, say, `AddressTest`. The class contains the standard `main` method and uses `new` to instantiate three `Address` objects:

```
class AddressTest
{  public static void main(String args[])
   {  Address friend = new Address();
      Address mother = new Address();
      Address father = new Address();
   }
}
```

This class should be saved in a file named `AddressTest.java` and can be compiled by typing:

```
javac AddressTest.java
```

The class refers to objects of type `Address` whose definition was stored in the file `Address.java`. Both files must be contained in the same directory so that `AddressTest` can find the definition of `Address`. To execute `AddressTest` we type:

```
java AddressTest
```

Nothing will appear on the screen because we have not implemented the methods in the `Address` class, nor do we specifically execute any method. Only the `main` method is called automatically by the Java Virtual Machine.

For the short time the `main` method was running we created three *real existing* objects of type `Address` in memory,[7] referred to by local variables named `friend`, `mother`, and `father`. Each contained three fields (`firstName`, `lastName`, and `email`) and had access to the shared method `showAddress`.

[7] A fourth object of type `AddressTest` is instantiated by the JVM via the command line `java AddressTest`.

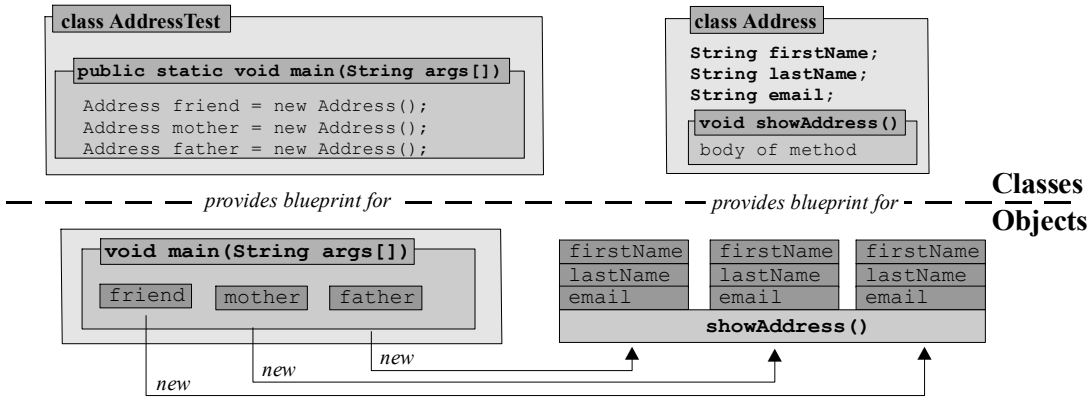


Figure 3.07: Objects instantiated from class

■

The `Address` and `Length` classes we designed so far have a major deficiency because there is no way to fill them with actual data. We don't know how to enter a real first and last name into an `Address` object, or how to set the `value` and `scale` of a `Length` object to anything useful. We need to explore more carefully what happens when an object is instantiated.

Constructor

A constructor is a special method of a class that is automatically executed when an object is instantiated from a class. Constructors do not have a return type and they have the same name as the class name. They use the syntax:

```
[modifier] ClassName(paramList)
{ /* implementation */ }
```

where `modifier` *is* `public`, `protected`, *or nothing. Constructors are used to initialize fields, allocate resources, and perform other housekeeping tasks. The input values to the constructor must match its parameter list in type and number during instantiation with* `new`.

Example 3.04: `Address` *class with constructor*

Add a constructor to our previous `Address` class and use it to enter data into our fields.

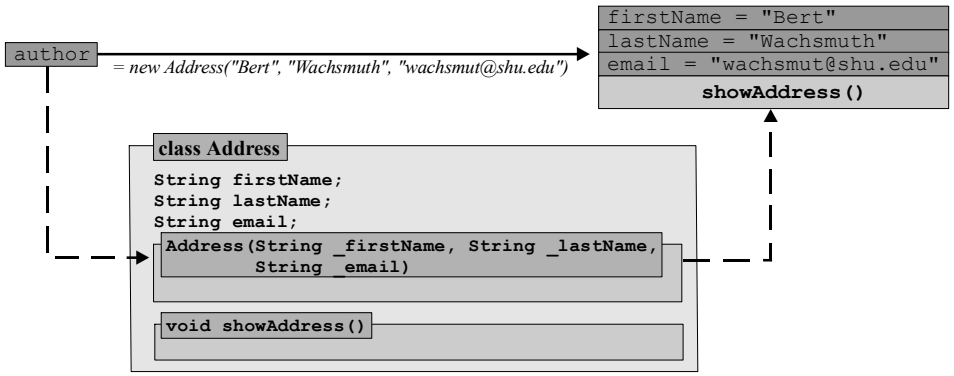
A constructor is a method defined in a class that has the same name as the class name and no explicit return type. To set the values for the `firstName`, `lastName`, and `email` fields of our class we use three input parameters of type `String`.

```
class Address
{  // Fields
   String firstName;
   String lastName;
   String email;
   // Constructor
   Address(String _firstName, String _lastName, String _email)
   {  firstName = _firstName;
      lastName = _lastName;
      email = _email;
   }
   // Methods
   void showAddress()
   {  /* implementation */ }
```

```
}
```

To enter actual data into an `Address` object we provide appropriate values to the constructor during the instantiation process, as in the following statement:

```
Address author = new Address("Bert", "Wachsmuth", "wachsmut@shu.edu");
```



When this line executes, the input values to the constructor `Address` are copied into the fields of the `Address` object named `author`. The `author` object can then manipulate these values as it sees fit.

■

> **Software Engineering Tip:** To easily recognize a constructor it should appear immediately after the fields in a class definition. Because of the scope rules its input parameters should be different from the field names. If the constructor initializes fields, use input parameters with the same name as the fields they initialize but preface them by an underscore character.[8] Constructors do not necessarily need input parameters.

Classes can be used inside other classes to define fields or local variables.

Example 3.05: Using `Address` *as field types for* `Parents` *class*

Write a complete program to store the addresses of your parents. The program should use the `Console` class introduced in section 2.4 to ask the user for the information.

Problem Analysis: We can use the `Address` class to store addresses. There are usually two parents with possibly different names so we need two objects of type `Address`. We create a `Parents` class to combine the addresses for both parents in one object.

Class Implementation: We have already created the `Address` class in example 3.04, including the constructor method to set the information for a particular address. The `Parents` class *has* two addresses, so that it will get two fields called `father` and `mother`. We add a constructor to `Parents` that asks the user for input to initialize the `father` and `mother` fields. The constructor obtains the necessary information from the user, so it does not need any input parameters.

```
class Parents
{  // Fields
   Address father;
```

[8] Other authors prefer using the same name as the fields that are initialized, but preface the field names inside the constructor by the special keyword `this`.

```
   Address mother;
   // Constructor
   Parents()
   {  System.out.print("Enter first name for father: ");
      String first = Console.readString();
      System.out.print("Enter last name for father: ");
      String last = Console.readString();
      System.out.print("Enter email address for father: ");
      String email = Console.readString();
      father = new Address(first, last, email);
      System.out.print("Enter first name for mother: ");
      first = Console.readString();
      System.out.print("Enter last name for mother: ");
      last = Console.readString();
      System.out.print("Enter email address for mother: ");
      email = Console.readString();
      mother = new Address(first, last, email);
   }
}
```

This class compiles fine, provided that the `Console.java` class from section 2.4 and the `Address.java` class from example 3.04 are in the same directory as `Parents.java`. But the class is not created well since its constructor contains code repetition. A better version of `Parents` uses *one* field of type *array* of `Address`. We can store two addresses by using an array of size 2 and we can use a loop to initialize the fields. Here is the improved version of `Parents`:

```
class Parents
{  // Fields
   String LABELS[] = {"father", "mother"};
   Address parents[] = new Address[2];
   // Constructor
   Parents()
   {  for (int i = 0; i < parents.length; i++)
      {  System.out.print("Enter first name for " + LABELS[i] + ": ");
         String first = Console.readString();
         System.out.print("Enter last name for " + LABELS[i] + ": ");
         String last = Console.readString();
         System.out.print("Enter email address for " + LABELS[i] + ": ");
         String email = Console.readString();
         parents[i] = new Address(first, last, email);
      }
   }
}
```

To finish our task, we need to create a third class that we call `ParentTest`. That class needs a standard `main` method so that it becomes executable. All it will do is instantiate an object of type `Parents`.

```
class ParentTest
{  public static void main(String args[])
   {  Parents parents = new Parents(); }
}
```

When we execute `ParentTest`, the following happens (see figure 3.08 for the output):

- the JVM calls the `main` method of `ParentTest`
- `main` instantiates one `Parent` object
- instantiation causes the `Parent` constructor to execute and ask for user input
- the `Parent` constructor instantiates two objects of type `Address`

- the `Address` constructor executes twice, once for each object, and initializes the fields

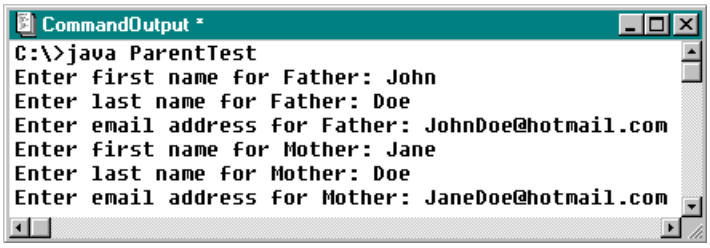


Figure 3.08: Executing the ParentTest class

It does not matter to `ParentTest` which version of the `Parents` class it uses. The user dialog and the end result of storing two addresses are the same.

■

Destructors and Garbage Collection

When an object is no longer needed the memory and other resources it occupies should be returned to the JVM for possible reallocation. Java provides an automatic mechanism called garbage collection to search out objects no longer in use and reallocate the resources they occupied.

Garbage Collection

> *Garbage collection is the process by which unused resources are returned to the JVM. Java's garbage collecting process is automatic and runs in the background.[9] When it executes, it identifies unused objects and returns the memory they occupied to the system.*
>
> *The garbage collection mechanism can be scheduled by calling the method* `System.gc()`*.[10] It will then execute at the earliest possible time, but not necessarily immediately.*

Reference objects that are no longer needed are automatically recycled. For example, local reference variables of a `private` method that is called only once are automatically marked for deletion when the method exits.

> **Software Engineering Tip:** Usually you need to do *nothing* to facilitate resource reallocation but you can specifically mark an object for recycling by setting its reference variable to `null`. For example, if you have stored a large amount of text in a `String` or `StringBuffer` field you may want to set it to `null` when the text is no longer needed to ensure that the memory used by the object can be recovered as soon as possible.

You can also call `System.gc()` manually to schedule garbage collection as soon as possible, but Java's automatic scheme is sophisticated enough to handle most situations on its own.

[9] It is a common source of errors in languages where garbage collection must be handled manually to release resources still needed or to cling to unneeded resources. Java's *automatic* garbage collection eliminates such errors.
[10] Java does provide other methods such as `System.runFinalization()` to fine-tune resource reallocation, but Java's automatic mechanisms will be sufficient for most projects.

Occasionally it is necessary to perform specific clean-up tasks when an object is destroyed. Java offers a *destructor* method for that purpose that executes automatically when an object is terminated.

Destructor

A destructor is a special method that is executed automatically at the time when Java's garbage collection mechanism is about to release the resources allocated to an object no longer in use. The destructor is called `finalize` *and is called automatically, never directly. Its syntax is:*

```
protected void finalize()
{  /* specific implementation */ }
```

Most classes have at least one custom-made constructor, but few also need a custom-made destructor.

Example 3.06:[11] Creating a destructor method

Add a destructor method to the `Parents` class from example 3.05.

The second version of the `Parents` class uses an array of two `Address` objects. To facilitate garbage collection, these objects can be set to `null` when the garbage collection mechanism tries to recycle a `Parent` object.

```
class Parents
{  // Fields
   String LABELS[] = {"father", "mother"};
   Address parents[] = new Address[2];
   // Constructor
   Parents()
   {  /* as before */ }
   // Destructor
   protected void finalize()
   {  super.finalize();
      for (int i = 0; i < parents.length; i++)
         parents[i] = null;
      parents = null;
   }
}
```

■

3.2. Class Access and Encapsulation

Now that we understand how to create our own classes we can discuss how to manipulate them, access their fields and methods, and how to restrict unwanted access if necessary.

Accessing Fields and Methods

Once an object has been instantiated, its fields and methods can be accessed in various ways:

[11] See example 10 in section 3.2. to see how and when a destructor executes.

Accessing Fields and Methods

Every object can access its own fields and methods by referring to them by name. Object A can access another object B if object A has a reference name pointing to object B:

- *It can use the reference name to refer to object B in its entirety.*
- *It can access the fields and methods of object B*[12] *using the dot operator and the syntax* `refNameForObjectB.memberOfObjectB`

Every object can refer to itself using the special keyword `this` *and to its own members using* `this.fieldOrMethodName`.

Here is a complete example using our previous `Address` classes.

Example 3.07: Calling on display feature of `Address` *and* `Parent` *classes*

In example 3.05 we created an `Address`, `Parents`, and `ParentTest` class to store two addresses. Modify the classes as necessary to store and display the addresses.

Problem Analysis: The `ParentTest` class uses a `Parents` and two `Address` objects to store the information. Both classes use constructors to initialize their fields. To obtain output, we delegate the work:

- `Address` contains a method `showAddress` to display the address it stores. We implement it to display an address in the form `"lastName, firstName (email)"`.
- `Parents` does not have a method to display its information so we add a method `showParents` to call on `showAddress` for both of its `Address` objects.
- The `main` method of `ParentsTest` can now call on `showParents` after instantiating a `Parents` object to display the information stored in that object.

Class Implementation: The `Address` class with the implemented `showAddress` method looks as follows:

```
class Address
{  // Fields
   String firstName;
   String lastName;
   String email;
   // Constructor
   Address(String _firstName, String _lastName, String _email)
   { /* as before */ }
   // Methods
   void showAddress()
   {  System.out.println(lastName + ", " + firstName + " (" + email + ")"); }
}
```

The `Parents` class can now call on the `showAddress` methods of the `Address` objects it contains:

```
class Parents
{  // Fields as before
   String LABELS[] = {"father", "mother"};
   Address parents[] = new Address[2];
   // Constructor
```

[12] Object B can use access modifiers to prohibited access to its fields and methods (see "Encapsulation" below).

```
   Parents()
   {  /* as before */ }
   void showParents()
   {  for (int i = 0; i < parents.length; i++)
      {  System.out.println("Address for " + LABELS[i]);
         parents[i].showAddress();
      }
   }
}
```

Finally the `ParentsTest` class can call `showParents` after constructing a `Parents` object:

```
class ParentTest
{  public static void main(String args[])
   {  Parents parents = new Parents();
      parents.showParents();
   }
}
```

Figure 3.09 shows the result of executing our improved classes:

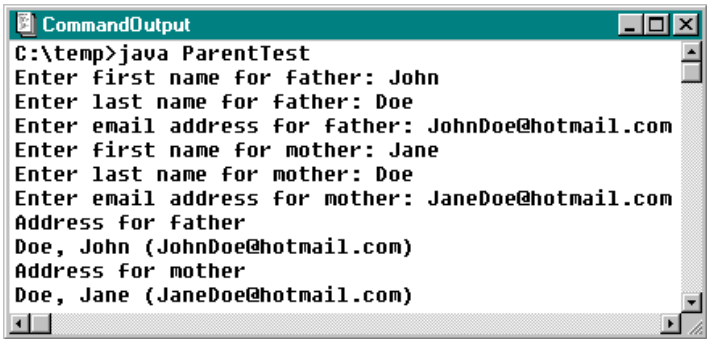


Figure 3.09: Executing the improved `ParentTest` *class to display the stored information*

■

Example 3.08: Implementing and testing the `Length` *class*

In example 3.02, we designed a `Length` class to convert lengths between feet and meter. Implement that class so that it can accomplish its task. Test your implementation using the `Console` class from section 2.4 to obtain user input.

Problem Analysis: We need to know the formulas for converting feet to meter and back. Recall that 1 meter = 3.2809 feet, or equivalently 1 foot = 1 / 3.2809 meter. Our conversion algorithm therefore looks as follows:

- To convert a `Length` object to feet:
  - if the length stored in the object is already of type feet, no conversion is necessary
  - if the length stored in the object is in meter, create a new length in feet by multiplying the original `value` by 3.2809
- To convert a `Length` object to meter:
  - if the length stored in the object is already of type meter, no conversion is necessary
  - if the length stored in the object is in feet, create a new length in meter by dividing the original `value` by 3.2809

Class Implementation: Recall the `Length` class as designed in example 3.02:

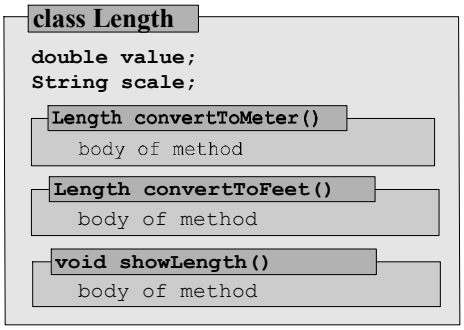


Figure 3.10: Representation of the `Length` *class from example 3.02.*

The class needs a constructor to set its fields. Since it has two fields the constructor needs two input parameters:

```
Length(double _value, String _scale)
{  value = _value;
   scale = _scale;
}
```

The conversion methods need to check the current value of `scale` to decide whether conversion is necessary, but they *must* return a `Length` as specified by their method headers. If no conversion is necessary, the current object can return itself using the keyword `this`. Otherwise a new `Length` object is constructed and returned, using the appropriate conversion formula as input to the constructor:

```
Length convertToFeet()
{  if (scale.equals("feet"))
      return this;
   else
      return new Length(value * 3.2809, "feet");
}
Length convertToMeter()
{  if (scale.equals("meter"))
      return this;
   else
      return new Length(value / 3.2809, "meter");
}
```

The `showLength` method displays a length in an appropriate format:

```
void showLength()
{  System.out.println(value + " " + scale); }
```

To create, convert, and display a length we can now use statements such as:

```
Length twoFeet = new Length("Feet", 2);
Length twoFeetInMeter = twoFeet().convertToMeter();
twoFeetInMeter.showLength();
```

But the `convertToFeet` method returns a new `Length` object, which has its *own* `showLength` method. We can call that method directly without storing the `Length` object in a separate variable:

```
Length twoFeet = new Length("Feet", 2);
twoFeet().convertToMeter().showLength();
```

An appropriate `LengthTest` class can now look as follows (the output is shown in figure 3.11):

```
public class LengthTest
{  public static void main(String args[])
   {  System.out.print("Enter a double value: ");
      double value = Console.readDouble();
      System.out.print("Enter a scale [meter or feet]: ");
      String scale = Console.readString();
      Length length = new Length(value, scale);
      System.out.print("Length in feet: ");
      length.convertToFeet().showLength();
      System.out.print("Length in meter: ");
      length.convertToMeter().showLength();
   }
}
```

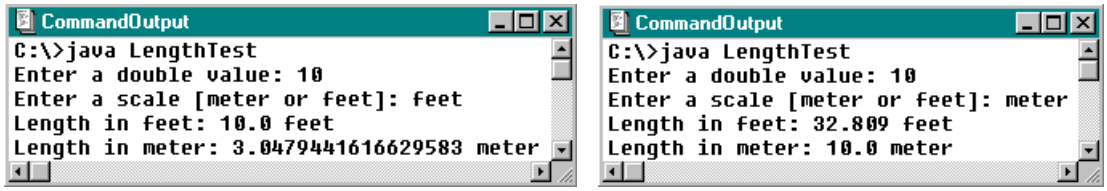
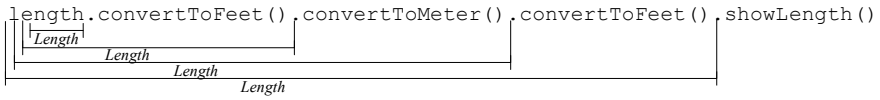


Figure 3.11: Executing `LengthTest` *to test the* `Length` *class*

We can even convert a length to feet, then to meter, then to feet, and display it in one statement:

```
length.convertToFeet().convertToMeter().convertToFeet().showLength();
```

because each method except `showLength` returns a `Length` object containing all methods specified in the `Length` class.



■

Private, Public, and Protected

It is often desirable to conceal certain aspects of a class from public view to hide unnecessary details or to prevent unauthorized access to data. In the `LengthTest` class of example 3.08 the `main` method asks the user to set the `scale` field of a `Length` object to `"feet"` or `"meter"` but if a user types `"foot"` the integrity of our class is destroyed and the convert mechanisms no longer work. There is no logical mistake in our class, but it should not allow unauthorized access to its data fields. The use of access modifiers can protect fields and methods through encapsulation.

Encapsulation using Private, Public, or Protected

> *Encapsulation is the process of allowing or disallowing access to a class or members of a class. Java provides the access modifiers* `public`, `protected`, `private`, *and none*[13] *to facilitate encapsulation, using the syntax:*

[13] Java provides the additional modifiers `static` (see "Static Fields and Methods" below), `abstract` and `final` (see section 3.4), `synchronized` (see section 5.3), and `native` and `volatile` (not used in this text).

```
[classModifier] ClassName
{  [memberModifier] fieldNames;
   [memberModifier] returnType methodName(inputParameters);
}
```

where `classModifier` *is either* `public` *or none and* `memberModifier` *is either* `public`, `private`, `protected`, *or none. The access granted by these modifiers is described in table 3.12.*

These access modifiers regulate access between objects and complement the scope rules of section 2.2.

Modifier	Access Granted
`public`	A public field, method, or class is accessible to every class
`protected`	A protected field or method is accessible to the class itself, subclasses, and all classes in the same package or directory.
`private`	A private field or method is accessible only to objects of the class in which it is defined.
none	A field, method, or class without modifiers is called friendly and is accessible to the class itself and to all classes in the same package or directory.[14]

Table 3.12: Meaning of `public`, `protected`, `private`, *or friendly access modifiers*

In this text we will only deal with projects where all classes reside in the same directory. In such classes there is little difference between friendly and `protected`, but we generally prefer to use `protected` over no modifier.

Example 3.09: Illustrating `private`, `public`, *and friendly modifiers*

To see how encapsulation works, create a class containing a `private`, `public`, and friendly field. Then create a second class with a standard `main` method, instantiate an object of the first class and try to access the fields of that object.

Here is the code for both classes (which must be saved in separate files):

```
1     public class AccessTest
2     {  // fields
3        private int privateNum = 1;
4        public int publicNum = 2.00;
5        int friendlyNum = 3;
6     }
7     public class Tester
8     {  // standard main method
9        public static void main(String args[])
10       {  AccessTest a = new AccessTest();
11          System.out.println(a.privateNum);
12          System.out.println(a.friendlyNum);
13          System.out.println(a.publicNum);
14       }
15    }
```

The results of compiling the class `Tester` are shown in figure 3.13.

[14] Fields and methods without specific access modifiers are also said to have `package` access.

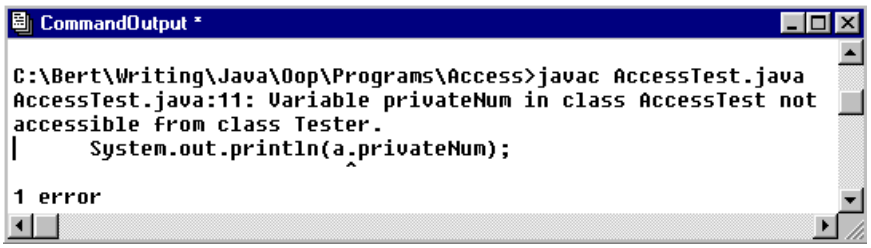
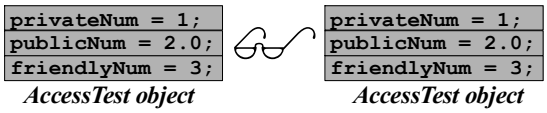
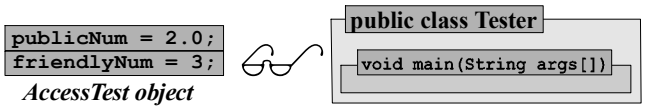


Figure 3.13: Error message when compiling `AccessTest`

The error message occurs because the class `Tester` cannot access the `private` field of object `a` of type `AccessTest`.



`AccessTest` *objects see each other*



`AccessTest` *object as seen from* `Tester` *class*

After removing line 11 from the `Tester` class, everything compiles fine and the output consists of the numbers `2` and `3`.

■

Having various modifiers to regulate access to class members, we need some guidance to determine when to use which one.

Software Engineering Tip: Access to class members should be as restrictive as feasible.

- All fields of an object should be `private` when possible and `protected` if necessary.
- Methods should be declared `private` if they can put an object in an invalid state or if they are not useful to other classes.
- Methods should only be declared `public` if they cannot produce any undesirable results.
- At least one method of a class should be `public`, `protected`, or friendly
- Classes and constructors should be either `public` or friendly. [15]

Example 3.10: Encapsulating the `Length` *class*

In our previous class `Length` we did not use access modifiers so all fields and methods are implicitly declared friendly. Redefine the class using the appropriate access modifiers.

We change the field declarations and method headers by preceding them with the appropriate access modifiers. Fields should be `private`, while useful methods should be `public`. Figure 3.14 shows the old and new class representations.

[15] The source code file name of a `public` class *must* be identical to the class name (plus the `.java` extension). `Public` classes have the advantage that they can be recompiled automatically by the Java compiler when necessary.

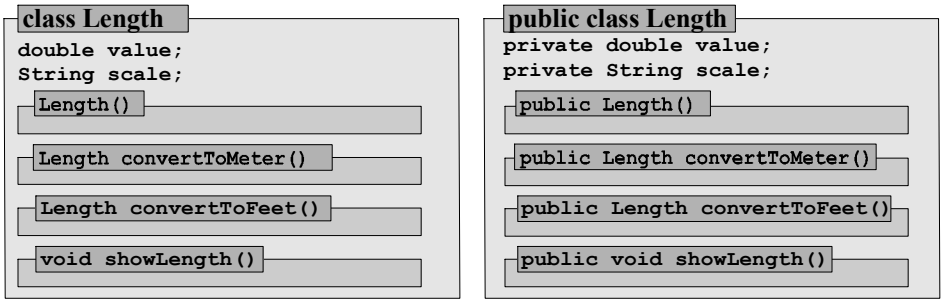


Figure 3.14: The `Length` *class without (left) and with (right) proper access modifiers*

The only way to display the values of a `Length` object in our new class is by using its `showLength` method. That ensures that a length will *always* be displayed in the proper format.

■

Marking fields as private does not mean a class is save from corruption. For the `Length` class, the statement `Length length = new Length(-10.0, "apples");` would compile and execute without problem, yet create an invalid length.

Example 3.11: Creating a safe `Length` *class*

Create a safe `Length` class, i.e. a class that has the same capabilities as the `Length` class from example 3.08 but prevents the user of the class from creating an invalid length.

Problem Analysis: We do not want to change the method headers so that our new `Length` class can be used as before. But we need to make sure that the `value` of a `Length` is never negative and `scale` is either `"meter"` or `"feet"`. The only method that can set these values is the constructor so we need to modify that constructor to ensure the safety of our class.

We cannot force a user to use proper input values to the constructor so the class needs an internal mechanism to mark a `Length` as invalid if improper input values were used during construction.

Class Implementation: To differentiate valid from invalid lengths we add a `private boolean` field `isValid` to the class. The constructor will set it to `true` if proper input values are used or to `false` otherwise. The remaining methods check `isValid` before they perform any action. Here is the new, safe `Length` class.

```
public class Length
{  private double value = 0;
   private String scale = "none";
   private boolean isValid = false;

   public Length(double _value, String _scale)
   {  value = _value;
      scale = _scale;
      if ((value >= 0) && ((scale.equals("meter")) || (scale.equals("feet"))))
         isValid = true;
   }
   public Length convertToFeet()
   {  if ((scale.equals("feet")) || (!isValid))
         return this;
      else
         return new Length(value * 3.2809, "feet");
   }
   public Length convertToMeter()
   {  if ((scale.equals("meter")) || (!isValid))
```

```
            return this;
         else
            return new Length(value / 3.2809, "meter");
      }
      public void showLength()
      {  if (isValid)
            System.out.println(value + " " + scale);
         else
            System.out.println("invalid length");
      }
   }
```

The constructor ensures that only `Length` objects with proper input values are marked as valid. Since `isValid` is a private field, its value can not change once an object is instantiated. Figure 3.15 shows several runs of the `LengthTest` class, the same class as in example 3.08:

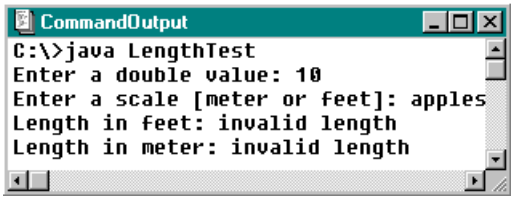


Figure 3.15(a): Entering an invalid `Length` *scale*

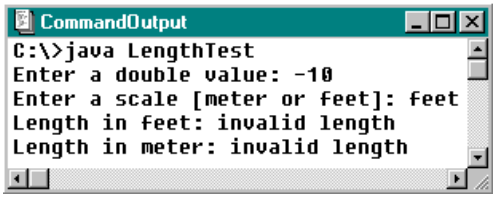


Figure3.15(b): Entering an invalid `Length` *value*

■

Marking fields as `private` is sometimes too restrictive, while declaring them as `protected` or `public` may leave a class vulnerable.

> **Software Engineering Tip:** To provide controlled access to `private` data fields of a class, create `public` *set* and *get* methods that act as a filter between the user of a class and any internal requirements the class needs to maintain.
>
> There is no standard for naming these methods, but the usual convention is to use the field name, capitalize the first letter, and preface it with the words `set` or `get`.

Example 3.12: A safe `Length` *class with set/get methods*

Add set and/or get methods to the `Length` class to allow controlled access to its private fields. Make sure the assumptions of the class remain true at all times.

Problem Analysis: A `Length` class might allow changing the value of an existing length but not its scale. It should produce its current value as a `double` and its scale as a `String`. A user may also want to know whether a given `Length` is valid.

Class Implementation: To satisfy these requirements we add the following set/get methods to Length:

- **`public double getValue()`**
  returns the value of a `Length` object as a `double`
- **`public String getScale()`**
  returns the scale of a `Length` object as a `String`
- **`public boolean isValid()`**

returns the current state of a `Length` object

- **`public void setValue(double newValue)`**
  changes the `value` of a `Length` object. If `newValue` is negative, sets the state of the object to invalid.

Here is the new `Length` class, including our set/get methods:

```
public class Length
{  private double value = 0;
   private String scale = "none";
   private boolean isValid = false;

   public Length(double _value, String _scale)
   { /* as before */ }
   public Length convertToFeet()
   { /* as before */ }
   public Length convertToMeter()
   { /* as before */ }
   public void showLength()
   { /* as before */ }
   public double getValue()
   {  return value; }
   public String getScale()
   {  return scale; }
   public boolean isValid()
   {  return isValid; }
   public void setValue(double newValue)
   {  if (newValue >= 0)
         value = newValue;
      else
         isValid = false;
   }
}
```

To test our new class, we use a simple test class as follows:

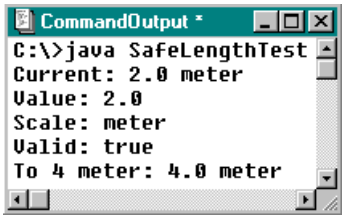


Figure 3.16: `SafeLengthTest`

```
public class SafeLengthTest
{  public static void main(String args[])
   {  Length length = new Length(2, "meter");
      System.out.print("Current: ");
      length.showLength();
      System.out.println("Value: " + length.getValue());
      System.out.println("Scale: " + length.getScale());
      System.out.println("Valid: " + length.isValid());
      System.out.print("To 4 meter: ");
      length.setValue(4);
      length.showLength();
   }
}
```

■

In addition to `private` fields many classes also have `private` methods for internal use. Such methods are frequently called *utility methods*.

Utility Method

> *A utility method is a* `private` *method of a class. The user of the class does not need to know its existence, but the method is needed by the class to perform its tasks and implement its* `public` *methods.*

In our `Length` class it seems clear that adding conversions other than from meter to feet and back will introduce additional methods and a user may get confused about method names.

Example 3.13: A safe `Length` *class with utility methods*

Change the methods `convertToMeter` and `convertToFeet` to `private` utility methods and introduce an appropriate public `convertTo` method to reduce the method names a user needs to memorize.

Starting with the previous definition of the `Length` class, we change the headers of the `convertToMeter` and `convertToFeet` methods so that they become `private` utility methods. We can then simplify these methods because they will be hidden from outside access and are no longer responsible for maintaining the integrity of the class. We introduce a new `convertTo` method that takes as input a `String` indicating the scale into which the length should be converted. That method will be `public` and must ensure that `convertToMeter` and `convertToFeet` are only called for valid conversions.

```
private Length convertToFeet()
{  if (scale.equals("feet"))
      return this;
   else
      return new Length(value * 3.2809, "feet");
}
private Length convertToMeter()
{  if (scale.equals("meter"))
      return this;
   else
      return new Length(value / 3.2809, "meter");
}
public Length convertTo(String newScale)
{  if ((isValid) && (newScale.equals("feet")))
      return convertToFeet();
   else if ((isValid) && (newScale.equals("meter")))
      return convertToMeter();
   else
      return new Length(-1.0, "invalid");
}
```

Any class that used to call on `convertToMeter` and `convertToFeet` will no longer compile. Our previous `LengthTest` classes therefore need to change to utilize `changeTo`.

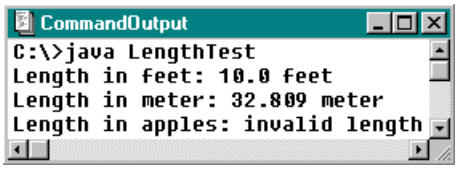


Figure 3.17: A new `LengthTest` *class*

```
public class LengthTest
{  public static void main(String args[])
   {  Length length = new Length(10, "feet");
      System.out.print("Length in feet: ");
      length.convertTo("feet").showLength();
      System.out.print("Length in meter: ");
      length.convertTo("meter").showLength();
      System.out.print("Length in apples: ");
      length.convertTo("apples").showLength();
   }
}
```

■

Software Engineering Tip: If a `public` method changes to `private` all classes that used to call it have to be redesigned. You should be careful when declaring a method as `public`, only methods

that are not likely to change should be `public`. Think of `public` methods and fields as a contract specifying the services your class offers. If you change them, your class breaks its contract.

Changing the implementation of a `public` method without changing the method header has no impact on other classes as long as the new implementation causes the same results. Changing `private` methods can by definition not impact on other classes.

Static Fields and Methods

Another modifier that regulates ownership and accessibility in conjunction with `public`, `private`, and `protected` is `static`. It allows you to define fields that are shared among all objects instantiated from a class and methods whose action depends only on its input parameters.

Class and Instance Fields

Class fields exist at the class level and are shared between all objects instantiated from a class. Class fields in Java are also called static fields and can be accessed as usual or without instantiating an object using the class name. Static fields are declared using the `static` *modifier:*

```
[public] class ClassName
{  [modifier] [static] type fieldName [= initialValue;]
   /* methods as needed */
}
```

A field declared without `static` *modifier is an instance field. Each instantiated object contains its own copy of instance fields.*

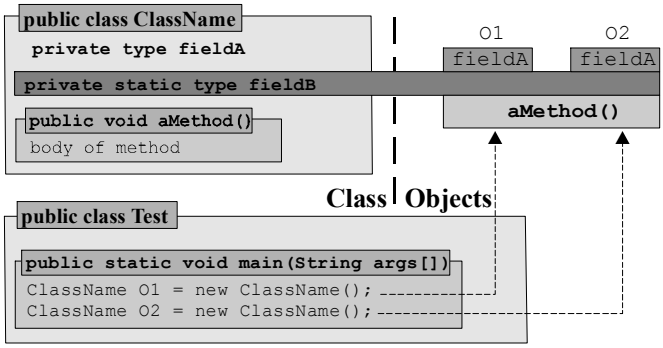


Figure 3.18: Instantiating objects with one static and one non-static field

A `static` field exists only once, no matter how many objects are instantiated. If one of these objects modifies a `static` field, all other instantiated objects will also receive that modification. Even if no objects are instantiated, a non-private `static` field can be accessed using the class name instead of an object name with the dot operator.

Example 3.14: Illustrating shared fields

A class `StaticTest` contains a `static` and non-`static` field as follows:

```
public class StaticTest
{  public int x = 1;
   public static int y = 1;
```

```
}
```

What is the output of the following `StaticTester` class?

```
1      public class StaticTester
2      {  public static void main(String args[])
3         {  StaticTest.y += 1;
4            StaticTest t1 = new StaticTest();
5            StaticTest t2 = new StaticTest();
6            t1.x += 1;
7            t1.y += 1;
8            t2.x += 2;
9            t2.y += 2;
10           System.out.println("T1: x = " + t1.x + ", y = " + t1.y);
11           System.out.println("T2: x = " + t2.x + ", y = " + t2.y);
12        }
13     }
```

When `StaticTester` executes, it causes the following action:

- In line 3 the `static` field `y` is incremented by `1`, using the class name instead of an object name. Since it was initialized to `1`, it now equals `2`.
- Lines 4 and 5 instantiate objects `t1` and `t2` of type `StaticTest`. Both objects have their own copy of the non-`static` field `x`, each with initial value `1`. The `static` field `y` is shared by `t1` and `t2` and has the value `2` because of line 3.
- Line 6 increments `t1.x` by `1` so it equals `2`.
- Line 7 increments `t1.y`, which is the same as `t2.y` and `StaticTest.y`, so it now has the value `3`.
- Line 8 adds `2` to the current value of `t2.x`, so `t2.x` will be `3`.
- In line 9 `t2.y` is again modified and will now have the value of `5`. Since `y` is `static`, `t1.y` will also be `5`.

The output produced by lines 10 and 11 is shown in figure 3.19.

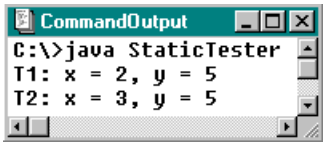


Figure 3.19: Running the `StaticTester` *class*

■

You can also have `static` methods in addition to `static` fields. Methods in Java are always shared between objects, but `static` methods can be called without instantiation.

Class Methods

Class methods are methods that exist at the class level and can be used without instantiating an object. Class methods are also called static methods and can be accessed as usual or without instantiating an object using the class name. They are declared using the `static` *modifier:*

```
[public] class ClassName
{  [modifier] [static] returnType methodName(inputParameters)
   { /* implementation using only static fields or input parameters */ }
}
```

A `static` *method can only refer to its input variables and to other* `static` *methods or fields.*[16]

Example 3.15: Creating shared methods

Define a class that provides formulas for computing the area of a rectangle and the volumes of a box and a sphere.

Problem Analysis: We need to collect the formulas for the various areas and volumes to compute:

- Rectangle with length l, width w: Area = l * w
- Box with length l, width w, and height h Volume = l * w * h
- Sphere with radius r Volume = 4/3 * π * r^2

Class Implementation: We implement a class `Geometry` to contain methods for computing the area and volumes. All computations depend only on the input parameters, so they can be `static`.

```
public class Geometry
{  public static double areaOfRectangle(double length, double width)
   {  return length * width; }
   public static double volumeOfBox(double length, double width, double height)
   {  return areaOfRectangle(length, width) * height; }
   public static double volumeOfSphere(double radius)
   {  return 4.0/3.0 * Math.PI * radius*radius*radius; }
}
```

The methods can now be used quickly and without instantiating any objects, as in:

```
System.out.println("Sphere volume, r = 3: " + Geometry.volumeOfSphere(3));
```

■

Software Engineering Tip: There are three frequent uses of the `static` modifier.

- Methods whose action only depends on the input parameters should be marked as `static` so they can be used quickly without instantiating objects. For example, the methods `Math.sqrt`, `Math.cos`, etc. are `static` methods, and `Math.PI`, `Math.E`, etc. are `static` fields.[17]
- The most common use of static fields is to define named constants using the modifiers `public static final`. Such a field is safe because its value can not be modified once it is initialized (because of `final`[18]) and memory is conserved because it exists only once (because of `static`).
- Static fields could be used to save memory. For example, if a class contains a large array of `double` as a field, marking it `static` will cause it to exist only once, regardless of the number of instantiated objects. You must be careful when marking `public` fields as `static` because any modification of the field will carry over to all instantiated objects.

The last example of this section uses a `static` field to count the number of instantiated objects.

[16] This explains why all fields and methods in chapters 1 and 2 were declared `static`. Everything was called from the standard `main` method without instantiation. Since that method is `static` it can only use other `static` fields and methods. If the `main` method instantiates objects they can use non-static class members.
[17] Other frequently used `static` methods include methods in the wrapper classes, methods in the `java.lang.System` package such as `System.out.println` and `System.gc`, and the standard `main` method of every Java program.
[18] Discussed in section 3.4.

Example 3.16: Using static fields to count instances of objects in memory

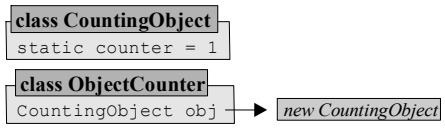
Compile and execute the classes below. The program uses the `Console` class introduced in section 2.4. Use the menu choices to create and delete objects and explain the output.

```
public class CountingObject
{  public static int counter = 0;

   public CountingObject()
   {  counter++;
      System.out.println("=> Created object # " + counter);
   }
   public void finalize()
   {  System.out.println("=> Dismissed object # " + counter);
      counter--;
   }
}

public class ObjectCounter
{  public static String showMenu()
   {  System.out.println("\n\t<C>reate object");
      System.out.println("\t<R>un garbage collector");
      System.out.println("\n\tE[x]it the program");
      System.out.print("\tEnter choice: ");
      return Console.readString();
   }
   public static void main(String args[])
   {  CountingObject obj = null;
      String choice = showMenu();
      while (!choice.equalsIgnoreCase("x"))
      {  if (choice.equalsIgnoreCase("c"))
            obj = new CountingObject();
         else if (choice.equalsIgnoreCase("r"))
            System.gc();
         choice = showMenu();
      }
   }
}
```

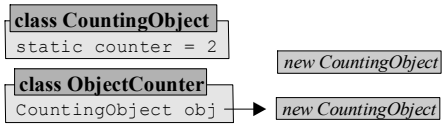
`CountingObject` contains a constructor and a destructor. The constructor increments a `static` variable `counter` and displays a message during instantiation. The destructor `finalize` is automatically called when the garbage collection mechanism reclaims memory occupied by an unused object of type `CountingObject`. It displays a message and decrements the value of `counter`.

The standard `main` method of `ObjectCounter` contains a reference variable `obj` of type `CountingObject`. You can choose to instantiate a new object or execute `System.gc` to schedule the garbage collection mechanism. Figure 3.20(a) shows the results of selecting `'c'` three times to create three objects and then `'r'` to schedule garbage collection.

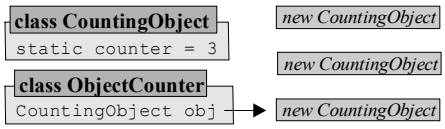
- There is only one reference variable `obj`. If `'c'` is pressed once, a new `CountingObject` is instantiated and referred to by `obj`.



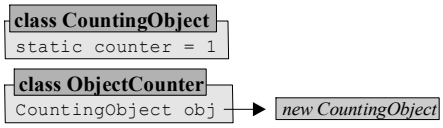
- If `'c'` is pressed again, a second `CountingObject` is instantiated and `obj` refers to it. The first instance of a `CountingObject` is still contained in memory but not referred to by any variable.



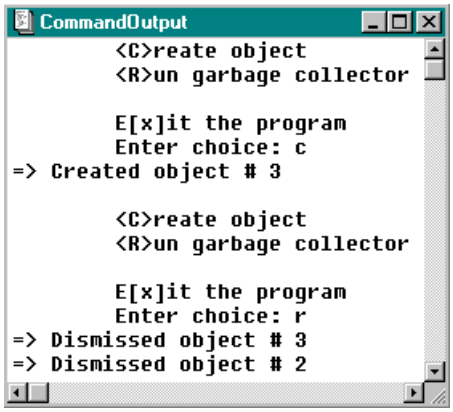
- If `'c'` is pressed again, a third `CountingObject` is created and referred to by `obj` while two `CountingObjects` are now without reference.



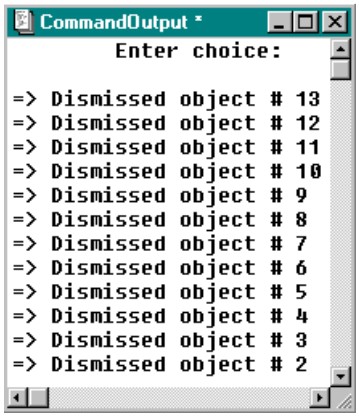
- If `'r'` is pressed the garbage collection mechanism can execute. It finds two instances of `CountingObject` in memory without reference and recycles them, executing the destructor once for each object.



The `static` field `counter` is a class field shared between all objects of type `CountingObject`. Because it is incremented in the constructor and decremented in the destructor it contains the number of object of type `CountingObject` in memory, whether they are referred to by `obj` or not.



(a): Pressing 'c' three times, then choosing 'r'



(b): Pressing 'c' thirteen times without pressing 'r'

Figure 3.20: Snapshots of two runs of `ObjectCounter`

Figure 3.20(b) shows a snapshot where `'c'` was pressed multiple times without choosing `'r'`. After a while the automatic garbage collection mechanism executes on its own, reclaiming the memory occupied by all `CountingObjects` that have no reference.

■

3.3. Overloading

When calling methods of a class, the input parameters must match the parameter list in the method header. It would be useful to allow for more flexibility and to be able to call one method with different input parameters. For example, we may want to construct a new `Address` using only a first and last name as input if the email address is unknown, or three input strings if they are known.

Overloading Methods

The idea of multiple definitions of one method is supported using overloading, an object-oriented concept that applies to all methods, including constructors (but not destructors).[19]

Overloading

Overloading is the ability to use one method to perform multiple tasks depending on the input values. Overloaded methods have the same method name and return type but the input parameters differ in type, in number, or both.

When an overloaded method is called, Java automatically chooses that definition where the input values match the parameter list in the method header.

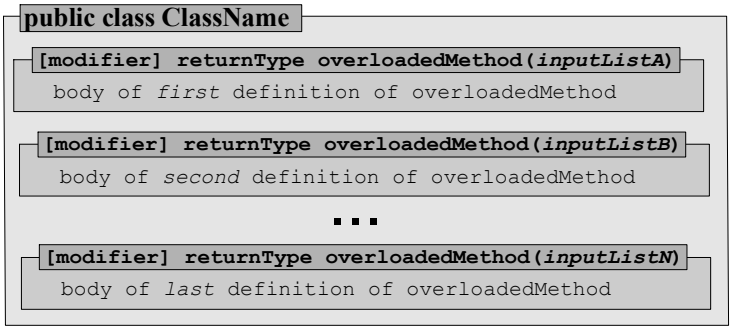


Figure 3.21: A class with an overloaded method named `overloadedMethod`

Example 3.17: Simple example using overloading

Suppose a class needs methods to double its input data of type `int` and `String` and display it on the screen. Use overloading to implement this feature.

Without overloading, the methods might look as follows:

```
public void doubleInt(int data)
{  System.out.println(2*data); }
public void doubleString(String data)
{  System.out.println(data + data); }
```

Using overloading, we define two methods called `doubleIt` with different input parameters:

```
public void doubleIt(int data)
{  System.out.println(2*data); }
public void doubleIt(String data)
{  System.out.println(data + data); }
```

If another method uses the statement `doubleIt(10)`, the compiler picks the first definition of `doubleIt`, displaying 20. If we use the statement `doubleIt("Pizza")`, the second definition is picked to display `"PizzaPizza"`.

■

Software Engineering Tip: Creating overloaded methods does not make the life of the programmer easier because he or she must still provide a separate method implementation for each version of the overloaded method. But using an overloaded method is much simpler,

[19] During garbage collection the only destructor called automatically is the one without input parameters.

because you only needs to remember one method that automatically accepts different input types. You should always consider overloading a `public` method to make it easier to use.

You *cannot* use overloading to create versions of a method that return different types. Use variations of the method name instead, such as `methodNameDouble` and `methodNameString`.

The prime example for the usefulness of overloading is the familiar `System.out.println` method.

***Example 3.18: Presumed definition of* `System.out.println`**

How does the definition of the `System.out.println` method look?

We have called `System.out.println` many times without every worrying about the type of input value we use. That method must be `public`, with `void` return type, and overloaded multiple times:

```
public void System.out.println(String s)
{  /* implementation */ }
public void System.out.println(double x)
{  /* implementation */ }
public void System.out.println(int i)
{  /* implementation */}
... etc, with implementations for each of the basic types
```

The programmer who created the method had to provide many different implementations, but we as users benefit from that work because we can call the method without worrying about the input type.

■

Example 3.19: Overloading static methods

Suppose we are working on a class that deals with geometric points in two and three dimensions. Write a complete method, or methods, that compute and return the distance to the origin for these points.

Problem Analysis: The distance to the origin is given by the formulas:

- if (x, y) is a point in 2-dimensional space, its distance to the origin is $\sqrt{x^2 + y^2}$
- if (x, y, z) is a point in 3-dimensional space, its distance to the origin is $\sqrt{x^2 + y^2 + z^2}$

Method Implementation: Instead of defining methods with different names, we use overloading. Since the formulas only depend on their input parameters, we can also mark the methods as `static`:

```
public static double distance(double x, double y)
{  return Math.sqrt(x*x + y*y); }
public static double distance(double x, double y, double z)
{  return Math.sqrt(x*x + y*y + z*z); }
```

The user only needs to remember that the method `distance` will compute the distance to the origin, regardless of the dimensions of the input points. Since the methods are `static`, they can be prefaced by the class name or an object name.

■

Overloading Constructors

The most common use of this mechanism is to overload the constructor of a class because it allows you to create multiple versions of an object depending on the input values used during instantiation.

Example 3.20: Overloading constructor for `Student` *class*

Create a `Student` class to store and display a student's name and GPA, if available. Provide multiple constructors to create students whose name or GPA is not known.

Problem Analysis: A flexible `Student` class should be able to create:

- an unknown student without name and GPA
- a student whose name is known but the GPA is not
- a student whose GPA is known but the name is not
- a student whose name and GPA is known

The output for a `Student` should display only the known information.

Class Implementation: The class needs two fields, one of type `String` to store the name and one of type `double` to store the GPA. To create different students, we overload the constructor. We need a constructor without input, with a `String` as input, a `double` as input, and a `String` and a `double` as input. Each constructor needs to initialize the fields, so they all need essentially the same code. We need a way to avoid code duplication:

> **Software Engineering Tip:** When you overload a constructor, provide one version that is as general as possible. All other versions can call this constructor using `this(parameterList)`, substituting default values for some or all of the parameters. This avoids code duplication and makes your code easier to manage. The call to the general constructor must be the first line in the remaining constructors.

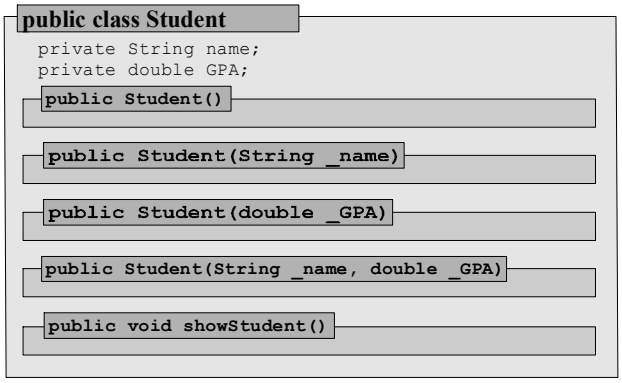


Figure 3.22: Representation of the `Student` *class*

```
public class Student
{  private String name;
   private double GPA;

   public Student()
   {  this("unknown", -1.0); }
   public Student(String _name)
```

```
   {  this(_name, -1.0); }
   public Student(double _GPA)
   {  this("unknown", _GPA); }
   public Student(String _name, double _GPA)
   {  name = _name;
      GPA  = _GPA;
   }
   public void showStudent()
   {  System.out.print("Student: " + name);
      if (GPA >= 0.0)
         System.out.println(" (GPA: " + GPA + ")");
      else
         System.out.println();
   }
}
```

The class is easy to use because of overloading as the following sample program illustrates.

```
public class StudentTest
{  public static void main(String args[])
   {  Student nobody = new Student();
      Student promise = new Student("Jane Smith");
      Student goodStudent = new Student(4.0);
      Student top = new Student("Jack Smith", 4.0);
      nobody.showStudent();
      promise.showStudent();
      goodStudent.showStudent();
      top.showStudent();
   }
}
```

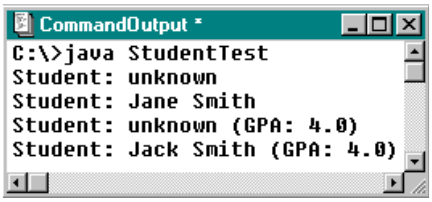


Figure 3.23: Output of `StudentTest`

■

Example 3.21: Using overloading to compute total course GPA

Create a complete program to store a list of course names you are taking, including a letter grade if available. The program should display the list of courses and your total GPA. Obtain user input using the `Console` class from section 2.4.

Problem Analysis: The program has two distinct responsibilities:

- It needs to handle course names with optional letter grades that need to be converted to numeric grades before computing an average. We model this situation with a `Course` class that has fields to store a name and a grade and methods to define a `Course`, to display a `Course`, and to convert a letter grade into a numeric grade.
- We need to obtain user input, store data, and compute an average. We use a second class `CourseList` for this. It has an array of `Course` objects as a field and methods to initialize the array and to display it with a computed total GPA.

Class Implementation: The fields for the `Course` class are of type `String`. To display a course we define a method `showCourse` with no input and `void` return type. To convert a letter grade to a numeric grade we declare a method `getNumericGrade` that returns a non-negative `double` corresponding to the letter grade or a negative number if no valid letter grade is available. A course may or may not have a grade associated with it, so we use an overloaded constructor. One version creates a course with a name, the second creates a course with a name and grade (see figure 3.24).

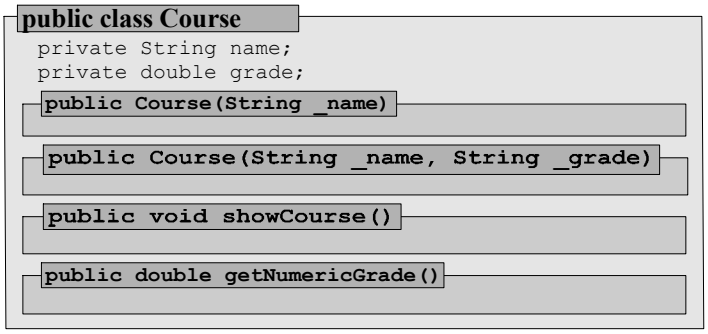


Figure 3.24: Representation of `Course` *class*

```
public class Course
{  private String name;
   private String grade;

   public Course(String _name)
   {  this(_name, "no grade"); }
   public Course(String _name, String _grade)
   {  name = _name;
      grade = _grade;
   }
   public void showCourse()
   {  System.out.println(name + " (Grade: " + grade+")"); }
   public double getNumericGrade()[20]
   {  String GRADES[] = {"A+", "A", "A-", "B+", "B", "B-",
                         "C+", "C", "C-", "D+", "D", "D-", "F"};
      double VALUES[] = {4.00, 4.0, 3.75, 3.25, 3.0, 2.75,
                         2.25, 2.0, 1.75, 1.25, 1.0, 0.75, 0};
      for (int i = 0; i < GRADES.length; i++)
         if (grade.equals(GRADES[i]))
            return VALUES[i];
      return -1.0;
   }
}
```

The `CourseList` class needs to perform two tasks: obtain user input to initialize the array of courses and display a list of courses with a computed GPA. Therefore it needs two methods, `getCourses` to create and initialize an array of `Course` objects and `showCourses` to display each course and to compute the total GPA. To create a complete program we also need a standard `main` method. User input is obtained using the `Console` class (see section 2.4).

```
public class CourseList
{  public static Course[] getCourses(int numCourses)
   {  Course courses[] = new Course[numCourses];
      for (int i = 0; i < courses.length; i++)
      {  System.out.print("Course description: ");
         String description = Console.readString();
         System.out.print("Grade available [y/n]: ");
         if (Console.readString().equalsIgnoreCase("y"))
         {  System.out.print("Enter grade [A,B,C,D,F]: ");
            courses[i] = new Course(description, Console.readString());
         }
         else
            courses[i] = new Course(description);
```

[20] A better implementation would define `GRADES` and `VALUES` as `static` fields. Currently these arrays are created every time the method is called. As `static` fields they would be created only once.

```
      }
    return courses;
    }
    public static void showCourses(Course[] courses)
    {  double sum = 0.0, num = 0.0;
       for (int i = 0; i < courses.length; i++)
       {  courses[i].showCourse();
          if (courses[i].getNumericGrade() >= 0.0)
          {  num++;
             sum += courses[i].getNumericGrade();
          }
       }
       System.out.println("GPA: " + (sum / num));
    }
    public static void main(String args[])
    {  System.out.print("Number of courses: ");
       Course courses[] = getCourses(Console.readInt());
       showCourses(courses);
    }
}
```

Figure 3.25 shows the results of executing the program. Grades such as I (for Incomplete) are stored in the course array and displayed but not used for computing the GPA.

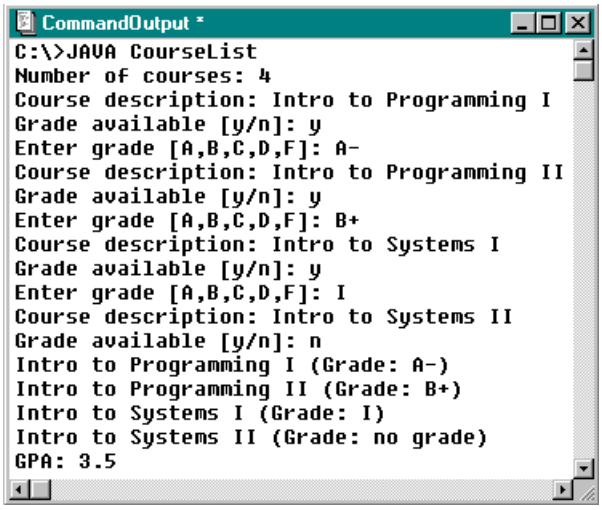


Figure 3.25: Results of executing the `CourseList` *class*

3.4. Inheritance

The great strength of object-oriented programming is that you can create classes that extend the capabilities of existing classes with little effort. This is especially important when creating programs with a graphical user interface (see chapter 4). Instead of recreating buttons and windows for every program, we can utilize "base classes" and extend their capabilities for a particular situation.

Creating Class Hierarchies

As an object-oriented programming language, Java allows you to create classes that form a hierarchy where one class can inherit properties of another class.

Inheritance

Inheritance is a language construct that allows for classes to be related to one another so that one class can inherit type and features of another class. The class or classes inherited from are called superclasses or ancestors. The inheriting classes are called subclasses or descendents. Inheritance is denoted by the keyword `extends`*, using the syntax:*

```
[modifier] class SubClassName extends ClassName
{ /* class implementation */ }
```

A subclass automatically possesses all of the non-private superclass methods and fields. It can access a superclass member by prefacing it with the keyword `super` *and the superclass constructor using* `super(parameter list)` *as the first line in the subclass constructor.*

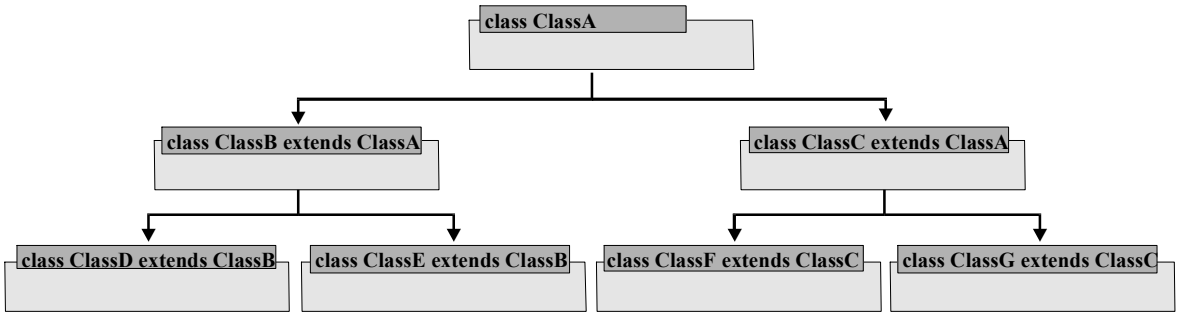


Figure 3.26: A sample inheritance hierarchy of classes

Figure 3.26 shows a possible hierarchy of seven classes related as follows:

- `ClassA` is the superclass (ancestor) for `ClassB`, `ClassC`, `ClassD`, `ClassE`, `ClassF`, and `ClassG`
- `ClassB` is the superclass (ancestor) for `ClassD` and `ClassE` and a subclass of `ClassA`
- `ClassC` is the superclass (ancestor) for `ClassF` and `ClassG` and a subclass of `ClassA`
- `ClassD` and `ClassE` are subclasses (descendents) of `ClassB` and `ClassA`
- `ClassF` and `ClassG` are subclasses (descendents) of `ClassC` and `ClassA`

Inheritance allows classes to utilize non-private fields and methods of their ancestors. It results in a reduced number of lines of code where features of multiple related classes can be modified by changing one superclass.

Example 3.22: Extending the `Address` *class*

Recall our `Address` class defined in example 3.01 with fields for the first name, last name, and email address. Define another class `PrivateAddress` as a subclass of `Address` that contains an additional field for a phone number.

The `Address` class in example 3.01 was defined as follows:[21]

[21] The fields and methods of `Address` and `PrivateAddress` are friendly (no access modifier). A better version of these classes would mark fields as `protected` and methods as `public`.

```
class Address
{  String firstName;
   String lastName;
   String email;

   void showAddress()
   {  /* implementation */ }
}
```

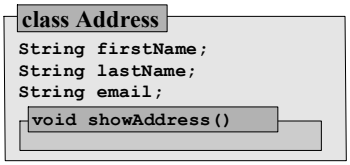


Figure 3.27: Representation of `Address`

To define `PrivateAddress` as a subclass of `Address` we use the `extends` keyword:

```
class PrivateAddress extends Address
{  /* implementation */ }
```

`PrivateAddress` *automatically* contains the non-private fields `firstName`, `lastName`, and `email`, which it inherits from `Address`. To store a phone number we only need to add one additional field:

```
class PrivateAddress extends Address
{   String phone; }
```

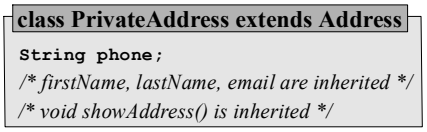


Figure 3.28: Representation of `PrivateAddress`

The new class also inherits the method `showAddress`, which does not know anything about the new field `phone`. We could add an extra method to display a "private" address[22]:

```
class PrivateAddress extends Address
{  String phone;

   void showPrivateAddress()
   {  /* implementation */ }
}
```

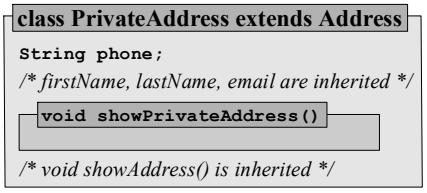


Figure 3.29: New `PrivateAddress`*class*

An object of type `PrivateAddress` now contains four fields (`firstName`, `lastName`, `email`, and `phone`) and two methods (`showAddress` and `showPrivateAddress`).

■

Software Engineering Tip: We have already seen that the phrase "has-a" implies that a class needs a field, and "does-a" suggests adding a method.

If a class "is-a" certain existing class, then the new class extends that class. Make sure that the features the subclass is interested in are not marked as `private` in the superclass, because `private` class members are not inherited. As an alternative, `private` fields can have `public` set/get methods in a superclass. Those methods *are* inherited and will work fine in the subclass.

[22] Section 3.4 defines overriding, a more convenient way to adjust superclass methods to a particular subclass.

Example 3.23: Creating `Shape` *classes using inheritance*

Suppose we need to write a program that deals with the area and perimeter of geometric shapes such as rectangles and circles, and possibly other shapes. Design these classes and create a test class to see if everything works.

Problem Analysis: Rather than creating different, unrelated classes for each of these objects we explore whether there is some relationship between them. The first sentence in the example already indicates that relationship: rectangles and circles are geometric shapes. We can use our "is-a, "has-a", and "does-a" tests to decide on the fields, methods, and hierarchies to use:

- a **Shape**:
  - "has-a" name, area, and perimeter
  - "does-a" display of its properties
- a **Rectangle**:
  - "is-a" shape
  - "has-a" width and height
  - "does-a" computation of area and perimeter
- a **Circle**:
  - "is-a" shape
  - "has-a" radius
  - "does-a" computation of area and perimeter

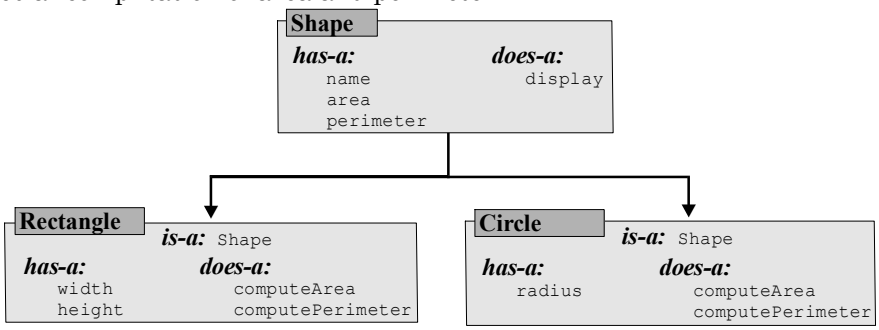


Figure 3.30: Hierarchy of `Shape` *classes*

Class Implementation: The class `Shape` serves as our superclass. It needs three fields (`name`, `area`, `perimeter`) and two methods (a constructor and a `display` method). To enable inheritance, none of them can be `private` so we mark the fields as `protected` and the methods as `public`.

```
public class Shape
{  protected String name;
   protected double area, perimeter;

   public Shape()
   {  name = "undetermined";
      area = perimeter = 0;
   }
   public void display()
   {  System.out.println("Name: " + name);
      System.out.println("Area: " + area);
      System.out.println("Perimeter: " + perimeter);
   }
}
```

Since we don't know the actual shape, we can not compute values for `area` and `perimeter`. `Rectangle` and a `Circle` are shapes, which we indicate by declaring them as subclasses of `Shape`. Both get constructors to set their own fields as well as methods to compute values for their inherited fields.

```
public class Rectangle extends Shape
{  protected double length, width;

   public Rectangle(double _length,
                    double _width)
   {  name   = "Rectangle";
      length = _length;
      width  = _width;
   }
   public void computeArea()
   {  area = length * width; }
   public void computePerimeter()
   {  perimeter = 2*(length + width); }
}
```

```
public class Circle extends Shape
{  protected double radius;

   public Circle(double _radius)
   {  name = "Circle";
      radius = _radius;
   }
   public void computeArea()
   {  area = Math.PI * radius * radius; }
   public void computePerimeter()
   {  perimeter = 2 *Math.PI * radius; }
}
```

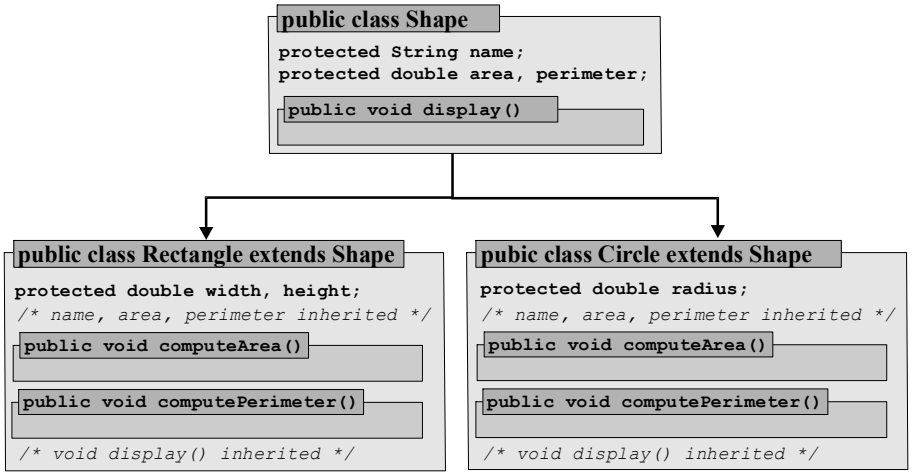


Figure 3.31: Inherited methods and fields for `Shape` *hierarchy*

To test these classes, here is a sample program[23]:

```
public class ShapeTester
{  public static void main(String args[])
   {  Shape s = new Shape();
      Rectangle r = new Rectangle(2.0, 3.0);
      Circle c = new Circle(4.0);
      r.computeArea();
      r.computePerimeter();
      c.computeArea();
      c.computePerimeter();
      r.display();
      c.display();
      s.display();
   }
}
```

```
CommandOutput
C:\>java ShapeTester
Name: Rectangle
Area: 6.0
Perimeter: 10.0
Name: Circle
Area: 50.26548245743669
Perimeter: 25.13274122872
Name: undetermined
Area: 0.0
Perimeter: 0.0
```

Figure 3.32: Running `ShapeTester` *class*

■

One advantage of inheritance is that it is easy to add new classes. We could easily add a `Square` to our collection of shapes using

[23] The classes `Shape`, `Rectangle`, and `Circles` must be saved as `Shape.java`, `Rectangles.java`, and `Circles.java`, respectively. `ShapeTester` must be saved as `ShapeTester.java`. All classes must be in the same directory.

```
public class Square extends Shape
{  /* implementation */ }
```

But a `Square` is a special rectangle so it should not extend `Shape` but `Rectangle`.

> **Software Engineering Tip:** The keyword `extends` might imply that a subclass is more general that a superclass. The contrary is true: subclasses are special cases of their superclasses. A picnic table extends a table in object-oriented terminology, even though a table is more general.[24]
>
> When creating a class hierarchy, use superclasses to combine or factor out common features of subclasses.

Before we can implement a `Square` class that extends `Rectangle` we need to clarify the impact of inheritance on instantiation.

Inheritance and Constructors

When a class is instantiated, its constructor is called automatically. When a subclass is instantiated, the constructors of the subclass *and* that of the superclass are called, even if the code does not include any calls to a constructor.

Implicit Call of Superclass Constructor

> *The constructor of a superclass is implicitly called with no input parameters when a subclass is instantiated, unless the subclass calls a superclass constructor explicitly as the first line of the subclass constructor. If the superclass does not contain a constructor with empty parameter list and the subclass does not explicitly call another superclass constructor, the compiler will not compile the subclass.*

Example 3.24: Illustrating implicit call of superclass constructor

Create a class called `SuperClass` that contains a constructor with a String input parameter that prints out a message. Create a second class called `SubClass` extending `SuperClass` that also contains a constructor printing another message. Try to instantiate an object of type `SubClass`. If an error occurs, fix it and compile the new classes.

This example consists of three classes: a `SuperClass`, a `SubClass`, and a simple `Test` class to instantiate the appropriate object(s):

```
public class SuperClass
{  public SuperClass(String msg)
   {  System.out.println("SuperClass constructor: " + msg); }
}

public class SubClass extends SuperClass
{  public SubClass()
   {  System.out.println("SubClass constructor"); }
}
```

[24] If `Square` extends `Rectangle` it inherits two fields (`width` and `height`) but a `Square` really needs only one. If `Rectangle` extends `Square`, `Rectangle` could not use a square's methods to compute `area` and `perimeter` and needs to create new methods. It is better to waste some (cheap) computer memory than programmer's (expensive) time.

```
public class Test
{  public static void  main(String args[])
   {  SubClass descendent = new SubClass(); }
}
```

Compiling the `Test` class produces the error message shown in figure 3.33.

Figure 3.33: Error message for incompatible superclass constructor call

When the `SubClass` is instantiated, its constructor will automatically call the `SuperClass` constructor with no arguments. Since the only constructor of the `SuperClass` requires a `String` as input, the `SubClass` can not be instantiated. We have to choices to fix this problem:

- We can overload the `SuperClass` to ensure that it contains a constructor with empty input parameter list:

```
public class SuperClass
{  public SuperClass()
   {  System.out.println("SuperClass constructor"); }
   public SuperClass(String msg)
   {  System.out.println("SuperClass constructor: " + msg); }
}
```

- We can change the `SubClass` by adding an explicitly call the original `SuperClass` constructor as the first statement of its constructor that providing a String as input:

```
public class SubClass extends SuperClass
{  public SubClass()
   {  super("Input message");
      System.out.println("SubClass constructor");
   }
}
```

Either solution compiles fine.

■

Software Engineering Tip: To avoid problems with implicit calls to a superclass constructor, use one of two remedies (or both):

- Every class extending another class should include an explicit call to one of the superclass constructors. That call must occur as the first line in the subclass constructor.
- Provide a constructor with no arguments for any classes that may become a superclass. If additional constructors are needed, use overloading to define them.

Now we can add a `Square` class extending `Rectangle` to our collection of shapes.

Example 3.25: Adding a `Square` to the `Shape` classes

Add a Square class to the collection of shapes from example 3.23.

We have already determined that `Square` should extend `Rectangle`. The new class is extremely short since it will inherit everything it needs except an appropriate constructor from `Rectangle`:

```
public class Square extends Rectangle
{  public Square(double _side)
   {  super(_side, _side);
      name = "Square";
   }
}
```

Calling the constructor of the `Rectangle` class initializes the fields correctly. The inherited methods `computeArea`, `computePerimeter`, and `display` automatically return the correct values. You should modify our previous `ShapeTester` class accordingly to test the new `Square` class.

■

A similar situation occurs when classes in a hierarchy contain destructors. Suppose a class serving as a superclass contains a destructor and another class extends it, containing a destructor of its own. Then the superclass destructor is implicitly called after the subclass destructor is finished. In this case few problems can arise because the default destructor does not require an input parameter.

Overriding

When a class extends another class, the subclass inherits all non-private fields and methods from the superclass. While that usually results in shorter code for the subclass without loosing any functionality, some of the superclass methods or fields may not appropriate for a subclass. In those cases a subclass can redefine or override them.

Overriding Methods and Fields

> *If a subclass defines a method that has the same method header as a method inherited from its superclass, the subclass method overrides, or redefines, the superclass method. The original superclass method is still available to the subclass using the keyword* `super` *and the dot operator. Methods that are overridden can not become more private than the superclass methods.*
>
> *If a subclass defines a field using the same name as the field inherited from its the superclass, the new variable overrides, or redefines, the superclass field. This is also known as hiding a superclass field. The original superclass field is still available to the subclass using the keyword* `super` *and the dot operator.*

Since you can use a superclass method while redefining it, overloading is common and can easily enhance a class.

Example 3.26: Overriding the `display` method in the `Shape` hierarchy

In the `Shape` hierarchy of example 3.23 the `Rectangle` and `Circle` classes both use the `display` method inherited from `Shape` but that method does not print out information particular to the new classes. Redefine the `display` method of the `Rectangle` class to also print out the values of the width and length fields.

The `Rectangle` class inherits a `display` method from `Shape`, which we override by using the same method header in `Rectangle` as in `Shape`. The `Shape`'s `display` method shows the `name`, `area`, and `perimeter`. To avoid code duplication, we use it in our overridden version and add code to show the `width` and `height` of a `Rectangle`.

```
public class Rectangle extends Shape
{  protected double length, width;

   public Rectangle(double _length, double _width)
   { /* as before */ }
   public void computeArea()
   { /* as before */ }
   public void computePerimeter()
   { /* as before */}
   public void display()
   {  super.display();
      System.out.println("Length: " + length);
      System.out.println("Width: " + width);
   }
}
```

No other class in the `Shape` hierarchy needs to change, not even the `ShapeTester` class. The new definition of the `display` method is used automatically (see figure 3.34).

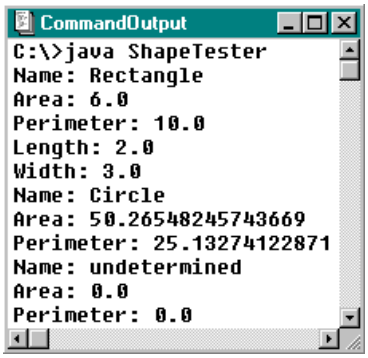


Figure 3.34: `Shape` *hierarchy with overridden* `display` *method*

■

Software Engineering Tip: As a general rule, overriding methods is as common as overloading, while overriding fields should be avoided.

- If a method of a superclass is not appropriate for a subclass, override it by adding a method with the *same* header to the subclass. If the functionality of the inherited method can still be used, avoid code duplication by calling the overridden method using the `super` keyword.
- If the input parameter list of the redefined method is *different* from that of the superclass, you have overloaded the superclass method, not overridden it.
- Be careful not to override a method by accident, you could inadvertently redefine an essential capability of a class. Make sure you know *all* inherited methods *before* defining your own.
- Avoid overriding fields, it makes your code difficult to understand. If many fields need to be hidden it might indicate that a subclass should not extend the superclass in the first place.

The overridden `display` method of example 3.26 is more appropriate for rectangles but still uses the superclass method. If the `display` method of the superclass `Shape` changes, the modifications will carry over to the `Rectangle`'s `display` method without touching the code of the `Rectangle` class.

Example 3.27: Illustrating usefulness of using `super` while overriding

Create an improved version of the `display` method for the `Shape` class of example 3.23. Verify that the new version will automatically be used in the `Rectangle` and `Circle` classes without recompiling them.[25]

Here is the new definition of our `Shape` class. Figure 3.35 shows that `Rectangle` and `Circle` utilize the new code even though their own code has not changed.

```
public class Shape
{  protected String name;
   protected double area, perimeter;

   public Shape()
   {  name = "undetermined";
      area = perimeter = 0;
   }
   public void display()
   {  System.out.println("******** GEOMETRIC SHAPE ******************");
      System.out.println(name+", area "+area+", perimeter "+perimeter);
   }
}
```

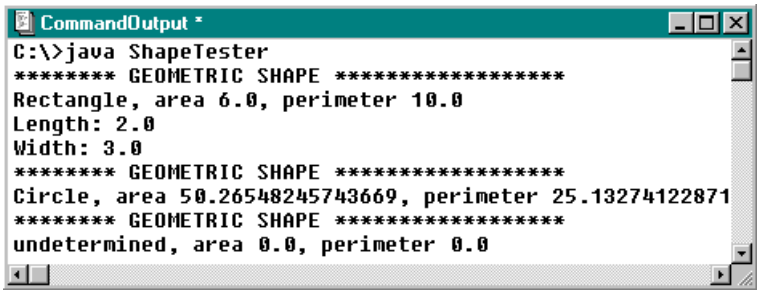


Figure 3.35: `Shape` *hierarchy with improved* `display` *method*

■

Abstract and Final

While inheritance is useful, there are situations where there clearly is a superclass but it is not possible to specify what a particular superclass method is supposed to accomplish. The superclass is simply too abstract to allow a concrete implementation of a method. In that case a class can define a method without providing an implementation by declaring it abstract.

Abstract Methods and Classes

An abstract method has a header but no implementation body. It must be declared as abstract by using the `abstract` *modifier. A class containing an abstract method is an abstract class and must be declared as such using the* `abstract` *modifier:*

```
[public] abstract class ClassName [extends Name]
{  [modifier] abstract returnType methodName(inputParameters); }
```

[25] The `Rectangle`'s `display` method can only utilize the improved version of `Shape`'s `display` method if it uses it. If it was overridden without calling `super.display` it would not notice the improvements.

A class that extends an abstract class inherits its abstract methods. If it does not implement them it also becomes an abstract class. Abstract classes can not instantiate objects.

Example 3.28: Abstract methods for `Shape` *classes*

The `Shape` class of example 3.23 has fields `area` and `perimeter` but no methods to compute these values because no information about the geometry of the shape is available. Define two `abstract` methods to compute the `area` and `perimeter` for `Shape`.

Defining abstract methods is easy, because they do not have an implementation body. When abstract methods are added to a class, it must be declared as an abstract class.

```
public abstract class Shape
{  protected String name;
   protected double area, perimeter;

   public Shape()
   {  /* as before */ }
   public void display()
   {  /* as before */ }
   public abstract void computeArea();
   public abstract void computePerimeter();
}
```

You can no longer instantiate any objects of type `Shape`. The `ShapeTester` class no longer compiles until the code instantiating a `Shape` object is removed.

■

A natural question is why abstract methods are useful. After all, they do not actually do anything in the above `Shape` class and they are immediately overridden in `Rectangle` and `Circle`.

Software Engineering Tip: Declaring abstract methods in abstract classes forces a subclass that wants to instantiate objects to implement the inherited abstract methods.

Declare a method as abstract if you cannot provide a method implementation but need to ensure that all subclasses implement that method.[26]

The next example will be longer and uses most of the techniques introduced so far.

Example 3.29: A menu-driven `CheckBook` *program using abstract methods*

Create a menu-driven checkbook program that can handle at least two types of transactions, checks and deposits. Use the `Console` class from section 2.4 for user input.

Problem Analysis: A checkbook program must be able to add checks and deposits, provide the current balance, and list all transactions. We create several classes to divide the responsibilities:

- We need a `Check` class that stores an amount and a description. It should return the amount as a negative value since money is spent.

[26] Ensuring that all classes in a hierarchy contain certain methods is important for polymorphism (see section 3.6).

- We need a `Deposit` class that stores an amount and a description. It should return the amount as a positive value since money is added to the account.
- We need a `CheckBook` class that stores `Check` and `Deposit` objects. It should let us add transactions such as checks and deposits, and it should display a list of all transactions together with the current balance. It should be initialized with a specific starting balance.
- We need a class containing a standard `main` method. That class should display a menu, ask the user for a choice, and act accordingly.

Software Engineering Tip: Programs are usually created to solve real-world problems.

- Identify real-world entities that make up a problem. Each is a candidate for a separate class.
- Use "has-a", "does-a", and "is-a" phrases to identify fields, methods, and relationships.
- Make each class as smart as possible, but each method as simple as possible.
- Avoid code duplication by using existing classes, inheritance, overloading, and overriding.
- One class – usually containing the standard `main` method - should act as the master class but the actual work should be performed by other classes.

A good object-oriented program works like a modern company. Each employee (class) is well trained and empowered to act as independently as possible, while the executive officer (master class), knowing the capabilities and limitations of the employees, develops the overall plan, distributes resources, directs responsibilities, and checks results but leaves the actual execution to the employees.[27]

Class Implementation: We need to create at least a `Check`, `Deposit`, `CheckBook`, and `CheckBookMain` class:

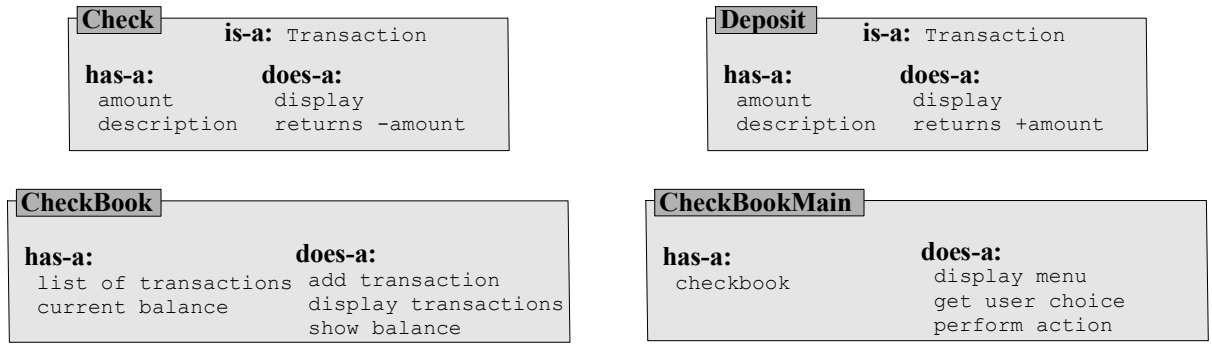


Figure 3.36: Suggested structure of classes for checkbook program

The `Deposit` and `Check` class need fields for the amount and description, and both need to return the amount, either as a positive or as a negative value. Instead of creating both classes separately we create a `Transaction` class and implement `Deposit` and `Check` as subclasses of `Transaction`. The `Transaction` class has fields for the amount and description and an additional field to indicate the type of transaction. It defines a method `getAmount`, but since it does not know the type of transaction, the method – and hence the class – is declared `abstract`. Checks and deposits contain user-definable fields, so `Transaction` provides a method `getDataFor` to interact with the user that `Check` and `Deposit` can use.

[27] Compare the *Case Study* section for additional details on designing object-oriented programs.

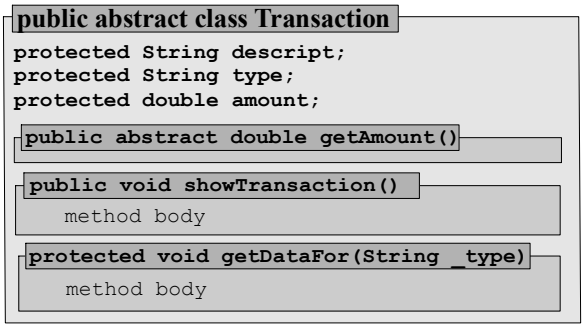


Figure 3.37: Representation of the Transaction class

```
public abstract class Transaction
{  protected String descript;
   protected String type;
   protected double amount;

   public abstract double getAmount();
   public void showTransaction()
   {  System.out.println("[ " +type +" ]\t" +getAmount() +"\t" +descript); }
   protected void getDataFor(String _type)
   {  type = _type;
      System.out.print("\n=> Description for " + type + ": ");
      descript = Console.readString();
      System.out.print("=> Enter amount: $");
      amount = Console.readDouble();
   }
}
```

Since `Check` and `Deposit` extend `Transaction`, they inherit its fields and methods. Their implementation is particularly easy.

```
public class Deposit extends Transaction
{  public Deposit()
   {  getDataFor("Deposit"); }
   public double getAmount()
   {  return amount; }
```

```
public class Check extends Transaction
{  public Check()
   {  getDataFor("Check"); }
   public double getAmount()

   {  return -amount; }
}
```

The class `CheckBook` needs to store the initial balance, which is neither a `Check` nor a `Deposit`. To model the initial balance, we create a class `Adjustment`. It extends `Transaction`, sets the type to `"Adjustment"`, and receives a description and an amount through its constructor. Like every class extending `Transaction` it must implement `getAmount`, which returns the positive amount since a new account must start with a positive initial balance. In an extended checkbook program the `Adjustment` class could be used for administrative transactions such as automatic interest payments or account corrections.

```
public class Adjustment extends Transaction
{  public Adjustment(String _descript, double _amount)
   {  type = "Adjustment";
      descript = _descript;
      amount = _amount;
   }
   public double getAmount()
   {  return amount; }
}
```

The class `CheckBook` needs to store a "list of transactions" and provide methods to add transactions to that list. We could create an array of `Check` and another array of `Deposit` objects, but `Check`, `Deposit`, and `Adjustment` all extend `Transaction`. Therefore they are of type `Transaction` in addition to their actual types. An array of `Transaction` objects can therefore store `Check`, `Deposit`, and `Adjustment` objects simultaneously. This does *not* violate our definition of an array since all of these objects *are* of the *same* type `Transaction`. Our next problem is that we do not know the size of the array of `Transaction` objects. Therefore we create an array of some large size, which is initially empty. New `Transaction` objects can be added to the array and a field `numTransactions` indicates the index of the last `Transaction` stored in the array. A final field `MAX_TRANSACTIONS` specifies the maximum number of transactions our program can handle.[28]

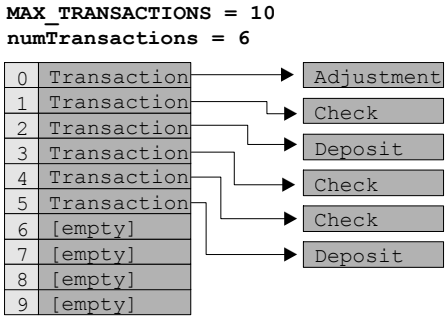


Figure 3.38: Transaction array with 10 possible and 6 actual transactions

```
public class CheckBook
{  private final static int MAX_TRANSACTIONS = 500;
   private Transaction transactions[] = new Transaction[MAX_TRANSACTIONS];
   private double balance = 0;
   private int numTransactions = 0;

   public CheckBook(double startBalance)
   {  add(new Adjustment("Starting balance", startBalance)); }
   public void add(Transaction aTransaction)
   {  if (numTransactions < MAX_TRANSACTIONS)
      {  transactions[numTransactions] = aTransaction;
         balance += aTransaction.getAmount();
         numTransactions++;
      }
      else
         System.out.println("\nMaximum number of transactions reached\n");
   }
   public void showCheckBook()
   {  System.out.println("\n=> Transaction:\t" + numTransactions);
      System.out.println("=> Balance:\t" + balance + "\n");
   }
   public void list()
   {  showCheckBook();
      for (int i = 0; i < numTransactions; i++)
         transactions[i].showTransaction();
   }
}
```

The `add` method of `CheckBook` adjusts the current `balance` by calling the `Transaction`'s `getAmount` method. A transaction is either of type `Check`, `Deposit`, or `Adjustment`, which each have their own

[28] Java provides a type called `Vector` to store objects of any type. A `Vector`'s size will adjust automatically to the number of objects stored. We introduce `Vector` in chapter 7 and a general list class in the *Case Study* section below.

`getAmount` method. The compiler always picks the method of the true type. For example, so if a `Transaction` is of type `Check`, the `Check`'s method `getAmount` will be used, if a `Transaction` is of type `Deposit`, the `Deposit`'s `getAmount` method will be used. Every object that extends `Transaction` has a `getAmount` method, because `Transaction` contains an abstract `getAmount` method that must be implement in subclasses.[29]

The final class, `CheckBookMain`, is easy because the actual work is handled by other classes. It instantiates a `CheckBook` object, displays a menu with choices, asks the user to make a selection, and acts accordingly by calling the appropriate methods of the `CheckBook` object. Figure 3.39 shows our collection of checkbook classes in action.

```
public class CheckBookMain
{  private static CheckBook checkBook = null;

   public static String showMenu()
   {  System.out.println("\n\t<C>\t to add a [C]heck");
      System.out.println("\t<D>\t to add a [D]eposit");
      System.out.println("\t<L>\t to [L]ist all transactions");
      System.out.println("\n\t<X>\t to e[X]it program");
      System.out.print("\n\tYour choice: ");
      return Console.readString();
   }
   public static void main(String args[])
   {  System.out.print("Starting balance: $");
      checkBook = new CheckBook(Console.readDouble());
      String choice = showMenu();
      while (!choice.equalsIgnoreCase("X"))
      {  if (choice.equalsIgnoreCase("C"))
            checkBook.add(new Check());
         else if (choice.equalsIgnoreCase("D"))
            checkBook.add(new Deposit());
         else if (choice.equalsIgnoreCase("L"))
            checkBook.list();
         choice = showMenu();
      }
   }
}
```

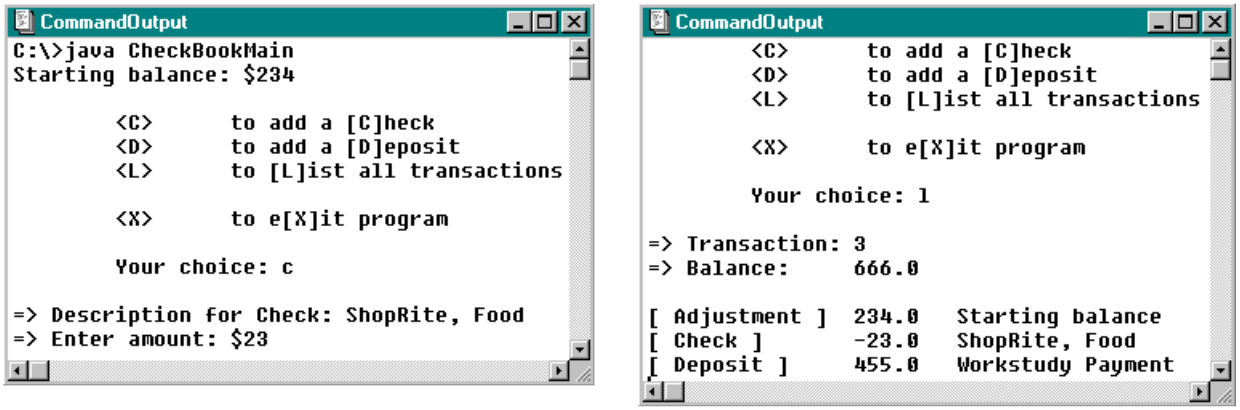


Figure 3.39: `CheckBookMain` *with initial balance of $234 and after adding a check and a deposit*

■

[29] This is an illustration of polymorphism, which will be discussed in detail in section 3.6.

Just as `abstract` can force a subclass to implement a method, Java provides a `final` modifier to restrict inheritance.

Final Methods and Classes

A final method is a method that cannot be overridden in a descendent class. A final class is a class that cannot be extended. Classes can contain final methods without being final themselves. Final classes or methods are declared using the `final` *modifier:*

```
[public] [final] className [extends ClassName]
{  [modifier] [final] returnType methodName(inputParameters)
   { /* implementation */ }
}
```

One reason for having final classes and/or methods is to protect the integrity of your class. For example, it could become a serious breach of the entire Java system if a programmer either by accident or by design could substitute his or her own `String` class in place of Java's `String` class. Therefore, the Java `String` class is a `final` class so that you can not derive any subclasses from it (for further security, the JVM performs special checks to be sure the `String` class is always authentic).

Software Engineering Tip: The most common use of the `final` modifier is to declare named constants, as discussed in section 1.5. Instead of using numeric values inside a program, declare them as `final static` fields at the beginning of a class and refer to them by name instead of by value. Because of `static` such fields exist only once and because of `final` they cannot change.

`Final` classes or methods should be used with caution because they disable inheritance and/or overriding, fundamental properties of object-oriented programming.

Example 3.30: Simple billing system using final fields and abstract methods

Create a billing system for students at *Java State University*. The rates that the university charges per credit hour are different for in-state and out-of-state students: $75 per credit hour for in-state students, $200 for out-of-state students. A bill for a student should contain the university's name, the student's name, the number of credit hours taken, and the total billing amount.

Problem Analysis: We create a `Student` class that serves as superclass for an `InStateStudent` and an `OutStateStudent` class. The `Student` class contains constants for the different rates, a field for the student's name, and a field for the credit hours taken. A method `showStudent` displays the student's name and total billing amount and a method `showSchoolName` displays the name of the university.

The `InStateStudent` and `OutStateStudent` classes inherit the fields and methods from `Student` and implement `showStudent` using the appropriate rate to compute the total billing amount.

Class Implementation: The first class to implement is the superclass `Student`. The fields for the rates are marked as `final` so that subclasses can not change them and as `static` so that they exist only once. The `showStudent` method can not determine the student's billing amount without knowing the type of student so it is marked as `abstract`. To ensure that the way the university's name is displayed cannot change, we mark `showSchoolName` as a `final` method.

```
public abstract class Student
```

```
{  protected final static double INSTATE_RATE = 75;
   protected final static double OUTSTATE_RATE = 200;
   protected String name;
   protected int hours;

   public abstract void showStudent();
   public final void showSchoolName()
   {  System.out.println("Java State University");
      System.out.println("**********************");
   }
}
```

The `InStateStudent` and `OutStateStudent` classes extend `Student` and must therefore implement the `showStudent` method. The values of the `name` and `hours` are set through the constructors.

```
public class InStateStudent
       extends Student
{  public InStateStudent(String _name,
                         int _hours)
   {  name = _name;
      hours = _hours;
   }
   public void showStudent()
   {  showSchoolName();
      System.out.println(name + " takes "
               + hours + " credits.");
      System.out.println("InState bill: "
               + hours * INSTATE_RATE);
   }
}
```

```
public class OutStateStudent
       extends Student
{  public OutStateStudent(String _name,
                          int _hours)
   {  name = _name;
      hours = _hours;
   }
   public void showStudent()
   {  showSchoolName();
      System.out.println(name + " takes "
               + hours + " credits.");
      System.out.println("OutState bill: "
               + hours * OUTSTATE_RATE);
   }
}
```

To test our billing classes we create a simple test class with a standard `main` method. Figure 3.40 shows its output.

```
public class BursarsOffice
{  public static void main(String args[])
   {  InStateStudent resident
         = new InStateStudent("John Doe", 24);
      OutStateStudent alien
         = new OutStateStudent("Joan Smith", 26);
      resident.showStudent();
      alien.showStudent();
   }
}
```

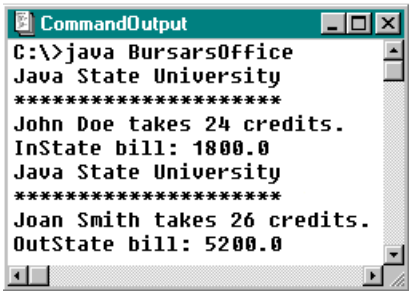


Figure 3.40: Testing the `Student` *classes*

■

3.5. The Basic Java Object

In the previous sections we used the keyword `extends` to create subclasses, but not every class was part of a hierarchy. As it turns out, every class declared in Java *automatically* extends a base class called `Object` and inherits all none-private fields and methods of that class. Therefore every class defined in Java has a minimal set of features as provided by `Object`.

The Object Class

Every class defined in Java is a subclass of a class called `Object`*, even if the keyword* `extends` *is not explicitly used. The* `Object` *class includes the following methods:*[30]

```
public class java.lang.Object
{   // constructor
    public Object();
    // selected methods
    public boolean equals(Object obj);
    protected void finalize();
    public final void notifyAll();
    public String toString();
    public final void wait(long timeout);
}
```

This idea of a base ancestor for every Java class has two advantages:

- Every class inherits the methods of the `Object` class. For example, every Java class has a method `toString` (compare example 3.31).
- Every class is of type `Object` in addition to its declared class type (compare example 3.33)

Example 3.31: `Object` as superclass for `Shape` hierarchy

In examples 3.23 we defined the subclasses `Rectangle` and `Circle` of the `Shape` class. Use the method `toString` to print out a string representation of a rectangle and a circle object, even though that method is not explicitly defined. What is the output?

Recall the definition and relationship of these classes:

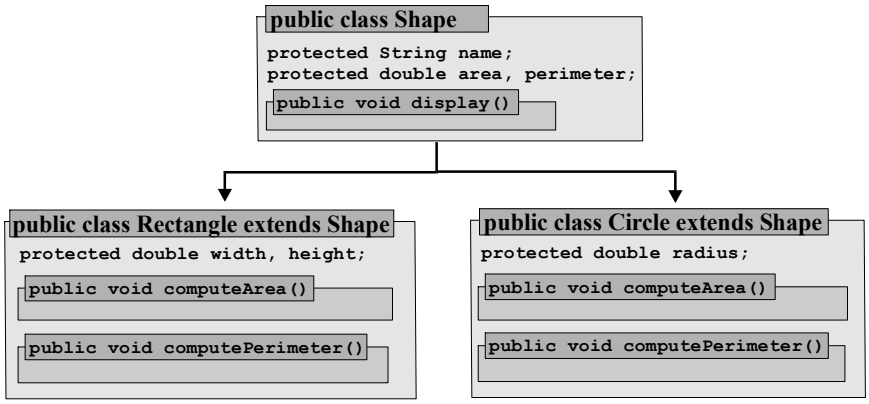


Figure 3.41: Explicitly declared `Shape` *hierarchy*

Figure 3.42 shows the apparent class hierarchy, where `Shape` seems the only superclass of `Rectangle` and `Circle`. But `Object` is an ancestor to all classes so that the true representation of the inheritance hierarchy is as shown in figure 3.41.

[30] For a complete definition of `Object`, check the Java API. Note that some methods are declared `final` and thus can not be overridden

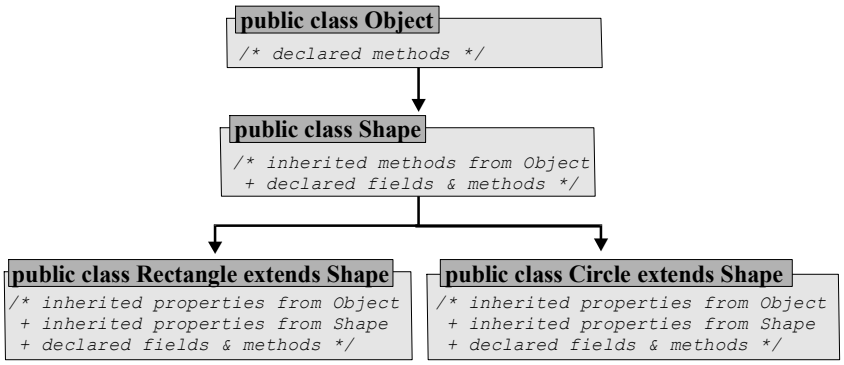


Figure 3.42: Actual `Shape` *hierarchy including base* `Object` *ancestor*

We can make use of the inherited `toString` method as follows:

```
public class ShapeTester
{  public static void main(String args[])
   {  Rectangle r = new Rectangle(2.0, 3.0);
      Circle c = new Circle(4.0);
      r.computeArea();  r.computePerimeter();
      c.computeArea();  c.computePerimeter();
      System.out.println(r.toString());
      System.out.println(c.toString());
   }
}
```

Figure 3.43: Output of `ShapeTester`

The `ShapeTester` class compiles since every Java class inherits `toString` from `Object`. Since that method is not specific to one of the `Shape` classes it prints the class name and memory location of the instantiated object (see figure 3.43).

■

> **Software Engineering Tip:** Every Java class implicitly extends `Object` and contains a `public toString` method. The `System.out.println` method uses `toString` *automatically* to convert an object to its `String` representation. Therefore, *every* class you design should override the `toString` method to provide useful information about itself. You can then display objects of your class using `System.out.println(objectName)` and `toString` is called automatically.

Example 3.32: Overriding the `toString` *method*

Redesign the `Shape` class of the above example to override the `toString` method. Modify `ShapeTester` to use the new method.

When we add a `toString` method, we also use it in `display` to avoid code duplication:[31]

```
public class Shape
{  protected String name;
   protected double area, perimeter;

   public String toString()
   {  return name + " [area = " + area + ", perimeter = " + perimeter + "]"; }
   public void display()
```

[31] The `display` method is now superfluous but we should leave it in our class because it was a `public` method. Removing `public` methods means that a class breaks its contract and other classes may no longer work.

```
   {  System.out.println(toString()); }
}
```

We can now modify `ShapeTester` to automatically use the overridden `toString` method. The output is shown in figure 3.44.

```
public class ShapeTester
{  // standard main methods
   public static void main(String args[])
   {  Rectangle r = new Rectangle(2.0, 3.0);
      Circle c = new Circle(4.0);
      r.computeArea();
      r.computePerimeter();
      c.computeArea();
      c.computePerimeter();
      System.out.println(r);
      System.out.println(c);
   }
}
```

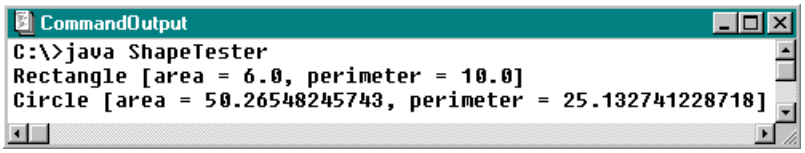


Figure 3.44: Output of `ShapeTester` *using overridden* `toString` *method*

■

The second advantage of extending an automatic base class is that every Java class is automatically of type `Object` in addition to its own type.

Example 3.33: Defining an array of `Object` *to store different types*

Define an array and insert a `Rectangle`, a `Circle`, a `double` and an `int`.

This seems impossible. An array must contain elements of the same type but `Rectangle` and `Circle` are objects of different types and `double` and `int` are not even objects. We could convert `double` and `int` into objects using wrapper classes (see section 2.5) but they are still objects of different types.

But every Java class extends `Object`, so all classes are also of type `Object`. Therefore we can create an array of `Object` types to store objects of any type. Elements can be retrieved from the array and typecast into their original class with the help of `instanceof`.

```
public class ObjectArray
{  public static void display(Object obj)
   {  if (obj instanceof Rectangle)
         System.out.println( ((Rectangle)obj).toString());
      else if (obj instanceof Circle)
         System.out.println( ((Circle)obj).toString());
      else if (obj instanceof Double)
         System.out.println( ((Double)obj).toString());
      else if (obj instanceof Integer)
         System.out.println( ((Integer)obj).toString());
   }
   public static void main(String args[])
   {  Object objects[] = new Object[4];
      objects[0] = new Rectangle();
      objects[1] = new Circle();
```

```
        objects[2] = new Double(1.0);
        objects[3] = new Integer(2);
        for (int i = 0; i < objects.length; i++)
           display(objects[i]);
     }
  }
```

The `display` method can be simplified significantly because every object contains a `toString` method, inherited or overridden, and there is no need to typecast the object to call this particular method. As example 3.32 illustrates, there is not even a need to call `toString` explicitly because `System.out.println` does that automatically.[32]

```
public static void display(Object obj)
{  System.out.println(obj); }
```

■

> **Software Engineering Tip:** If you declare an array as an array of `Object`, it can include any object, because all classes implicitly extend `Object`. To include basic types, use the wrapper classes to convert them into objects first. To use objects from the array you can typecast them back into their actual types. To use methods common to all objects, you do not need typecasting.
>
> To store *related* objects in an array, create a superclass and declare the objects in the array of that superclass type. If you declare an abstract method in the superclass, all objects extending the superclass must implement the method and you can use it without typecasting.

If you know the number of objects to store ahead of time, an array works well. But in many situations the number of objects to store depends on user action or other parameters that change during the course of an executing program. An array may not be useful in those cases.

Example 3.34: A general `List` *class*

Create a class called `List` that can store objects of any type such that you can add and remove objects any time.[33] Document and test the class.

Problem Analysis: An array of `Object` can store objects of any type, but it has a fixed size once it is initialized. What we need is a structure where objects can be added and removed *after* instantiation.[34] We therefore create a class that contains a large array of `Object` as a field. Initially the array is empty but with the help of `add` and `delete` methods objects can be inserted into and removed from the array as necessary. The class supports the following operations:

- add an element
- delete specified element
- retrieve a particular element without removing it
- return the current number of elements stored
- find out whether there is room for additional elements to store
- a `toString` method that should be overwritten in every class

[32] The `toString` method used is the one contained in the definition of the actual class of the object., or the inherited one if the class does not override `toString` . That is the underlying principle of polymorphism, explored below.
[33] Java has several build-in classes to store objects dynamically. The most commonly used is the `Vector` class of the `java.util` package. We will describe these classes in detail in chapter 7.
[34] In example 11, section 3.4, we have used that approach (without deletion) to store financial transactions.

Class Implementation: Our class contains an array of `Object` elements that has a fixed size but contains no objects. An integer field `numItems` contains at any time the current number of objects stored in the array. Every time an object is added to the class, it is inserted in the array and `numItems` is incremented by one. That means that our class has upper limit of objects it can store. To make our class more flexible, we set that upper limit in the constructor. We also provide a constructor without parameters to use a default upper limit. Here is the class skeleton:

```
public class List
{  private int maxItems  = 100;
   private int numItems  = 0;
   private Object[] list = null;

   public List()
   {  list = new Object[maxItems]; }
   public List(int _maxItems)
   {  maxItems = _maxItems;
      list = new Object[maxItems];
   }
   public void add(Object obj)
   { /* implemented below */ }
   public void delete(int pos)
   { /* implemented below */ }
   public Object get(int pos)
   {  return list[pos];  }
   public int getSize()
   {  return numItems; }
   public boolean isFull()
   {  return (numItems >= maxItems); }
   public String toString()
   { /* implemented below */ }
}
```

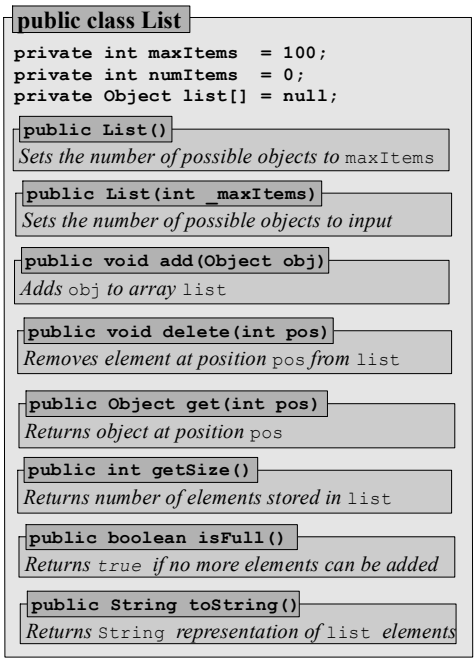
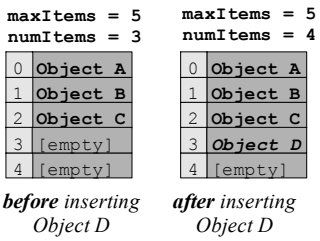


Figure 3.45: Representation of `List` *class*

The constructors, `getSize`, `isFull`, and `get` methods are already implemented. The `add` method appends the input object to the end of the array `list` and increments `numItems` if there is room.

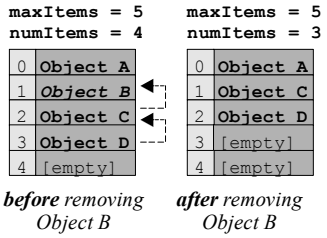
```
public void add(Object obj)
{  /* ASSUMES that there is room to add obj,
      i.e. assumes that numItems < maxItems. */
   list[numItems] = obj;
   numItems++;
}
```



To remove the element at position `pos`, the element at `pos+1` is moved into position `pos`, the element at `pos+2` is moved into position `pos+1`, etc, until all elements above `pos` have dropped down by one. Then `numItems` is decremented. That overwrites the element at position `pos` and frees one position at the end of the array.[35]

[35] If the element at position `numItems` is removed, the `for` loop does not execute but `numItems` is still decremented. The previously last element is no longer accessible but is still contained in the array. The situation is similar when a file is deleted from a disk. It is not physically erased, but its entry in a table of contents called FAT (file allocation table) is removed. That makes the file inaccessible, but its content is still present until overwritten.

```
public void delete(int pos)
{  /* ASSUMES that pos is between 0 and numItems */
   for (int i = pos+1; i < numItems; i++)
         list[i-1] = list[i];
   numItems--;
}
```

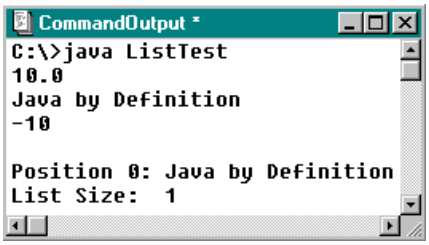


The method `toString` loops through all objects in the array and calls their inherited or overwritten `toString` method to create a `String` representation of the list.

```
public String toString()
{  String s = new String();
   for (int i = 0; i < numItems; i++)
      s += "\n" + list[i].toString();
   return s + "\n";
}
```

We use a simple test program to see if everything works properly:

```
public class ListTest
{  public static void main(String args[])
   {  List list = new List();
      list.add(new Double(10.0));
      list.add(new String("Java by Definition"));
      list.add(new Integer(-10));
      System.out.println(list);
      System.out.println("Position 0: " +
                         list.get(1));
      list.delete(2);
      list.delete(0);
      System.out.println("List Size:  " +
                         list.getSize());
   }
}
```



It remains to document our class. We embed documentation comments so that we can use the `javadoc` tool to create the documentation as described in section 2.4. The most important part of this documentation is the assumptions that must be satisfied to use `add` and `delete`.

```
/** Class to store objects dynamically up to a fixed limit. Supports basic operations
    <code>add</code>, <code>delete</code>, and <code>get</code>. <I>Note the
    instructions below before using these operations.</I>

    @author: Bert G. Wachsmuth
    @version: 2000/01/22
*/
public class List
{  /** Creates empty list. The default maximum size is 100. */
   public List()
   {  /* implementation as above */ }
   /** Creates empty list of a max size specified by <code>_maxItems</code>.*/
   public List(int _maxItems)
   {  /* implementation as above */ }
   /** Adds <code>obj</code> to the list. Before using <code>add</code> you should use
       <code>isFull</code> to ensure that the list is not full. */
   public void add(Object obj)
   {  /* implementation as above */ }
   /** Removes object at position <code>pos</code> from the list. Ensure that
```

```
      <code>pos</code> is between 0 and <code>getSize</code>. */
   public void delete(int pos)
   {  /* implementation as above */ }
   /** Returns element at position <code>pos</code>. Ensure that <code>pos</code> is
       between 0 and <code>getSize</code>. */
   public Object get(int pos)
   {  /* implementation as above */ }
   /** Returns the number of objects currently stored in the list. */
   public int getSize()
   {  /* implementation as above */ }
   /** Returns <code>true</code> if no more objects can be added to the list.*/
   public boolean isFull()
   {  /* implementation as above */ }
   /** Returns a simple string representation of the elements in the list. */
   public String toString()
   {  /* implementation as above */ }
}
```

■

This class provides a generally useful service, it can work with every type of object, and it does not depend on any other classes. It is our *second universally useful* class (the first is the `Console` class).

3.6. Interfaces and Polymorphism

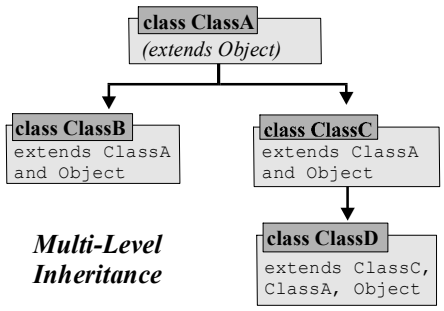
We covered the object-oriented concepts *classes*, *instances*, *encapsulation*, *overloading*, and *inheritance* in previous sections. This section presents the remaining topic of *polymorphism*, including a discussion of interfaces and multiple inheritance.

Multiple Inheritance and Interfaces

Class hierarchies can be many levels deep. Subclasses extending a superclass can in turn be extended without limit, but each class can have at most one *immediate* superclass.

Multi-Level and Multiple Inheritance

> *Multi-level inheritance occurs when a subclass of a superclass is itself extended to another class. Multi-level inheritance is supported in Java and is useful to create efficient class hierarchies.*
>
> *Multiple inheritance is a concept where a class simultaneously extends more than one superclass at the same level of a hierarchy. Multiple inheritance is not supported in Java.*



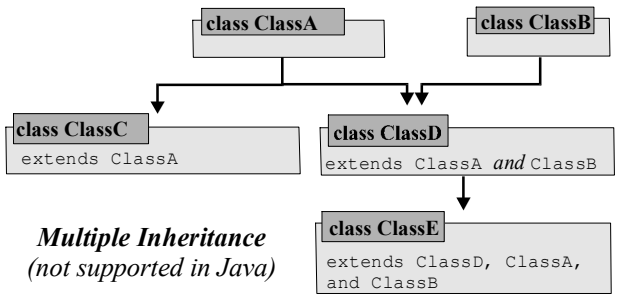


Figure 3.46: Multi-level versus multiple (not supported) inheritance

Example 3.35: Inheritance type of `Shape` *hierarchy*

In example 3.25 we added a `Square` class to our collection of geometric shapes. Did we use multiple inheritance or multi-level inheritance?

Recall that we defined `Square` extending `Rectangle`, which in turn extended Shape (which implicitly extends `Object`)[36]:

```
public class Square extends Rectangle
{  public Square(double _side)
   {  super(_side, _side);
      name = "Square";
   }
}
```

The `Square` class directly extends one class, which in turn extends one other class. Hence, we used multi-level inheritance, not multiple inheritance (after all, the class compiled so it could not have been multiple inheritance).

■

Example 3.36: Implicit call of superclass constructor in multi-level hierarchies

If we create a multi-level hierarchy three levels deep where each class has a constructor, are all superclass constructors invoked when the lowest level class is instantiated, or only the immediate superclass constructor?

We define three classes, together with a `Test` class containing the standard `main` method:

```
public class SuperSuper
{  public SuperSuper()
   {  System.out.println("SuperSuper Class"); }
}
public class Super extends SuperSuper
{  public Super()
   {  System.out.println("Super Class"); }
}
public class Sub extends Super
{  public Sub()
   {  System.out.println("SubClass"); }
}

public class Test
{  public static void main(String args[])
   {  Sub c = new Sub(); }
}
```

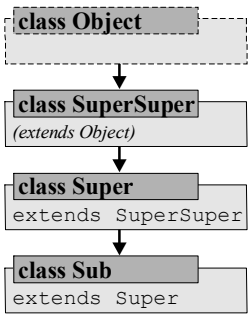


Figure 3.47: Multi-level hierarchy

Compiling and executing `Test` will answer the question, which is left as an exercise.

■

Multiple inheritance is not allowed in Java, but an interface mechanism can be used instead.

[36] If `Square` did not include an explicit call to its superclass constructor with two `double` as parameters, it would not compile because the superclass constructor would be called implicitly with no arguments.

Interfaces

An interface is a collection of abstract methods and final fields. It serves as a contract or promise, ensuring that every class implementing it contains the methods of that interface. Methods in an interface are automatically `abstract`*, fields are automatically* `final`*. An interface cannot be used to instantiate objects, and it is not a class. Interfaces should be* `public` *and can extend other interfaces.*

```
public interface InterfaceName [extends InterfaceList]
```

A class can use one or more interfaces via the keyword `implements`*. It inherits the abstract methods and final fields of the interface(s) and is considered to be of the type of the interface(s) in addition to its own type. It must implement the inherited methods or be declared* `abstract`*.*

```
[public] class ClassName [extends Class] implements Interface1 [,Interface2]
```

Example 3.37: Interfaces as a new type

Suppose an interface `Sortable` is defined as

```
public interface Sortable
{  /* details omitted */ }
```

and we change the definition of the `Square` class (see example 3.25) to:

```
public class Square extends Rectangle implements Sortable
{  /* details omitted */ }
```

What are the types of an object `square` instantiated via

```
Square square = new Square()
```

Obviously, `square` is of type `Square` since it is instantiated from that class. Since `Square` extends `Rectangle`, it is also of type `Rectangle`. That class in turn extends `Shape`, so `square` is also of type `Shape`. `Square` implements `Sortable`, so `square` is of type `Sortable` as well. Finally, every class extends `Object`, so `square` is also of type `Object`. Hence the types of `square` are:

`Square`, `Rectangle`, `Shape`, `Object`, and `Sortable`

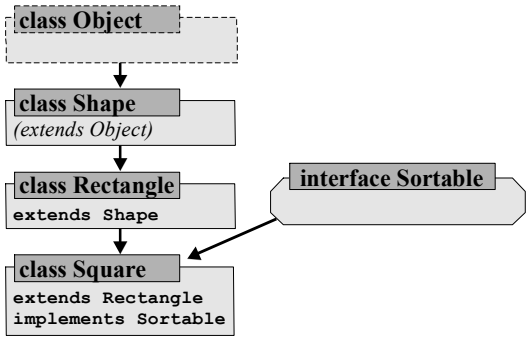


Figure 3.48: Types of `Square` *object*

■

Interfaces are conceptually different from classes but work in conjunction with them to provide "place-holder" methods or types.

Software Engineering Tip: The most common use of interfaces involves three steps:

1. The need to perform an abstract action or possess a conceptual property. That action or property is specified in an interface, which symbolizes a placeholder for a concrete type.
2. A specific algorithm or plan that relies on the abstract action or property in (1). That algorithm or plan is implemented in a class using objects defined in the interface.
3. One or more classes that specify a particular incarnation of the action or property in (1) and use the plan or algorithm provided in (2)

Use these steps if you have a certain plan or algorithm that works great if it could use objects with property X, but property X is abstract and varies for different groups of objects.

Example 3.38: Sorting arrays using a `Sorter` *class and* `Sortable` *interface*

Create a reusable `Sorter` class that can sort arrays of any type. Use that class to sort an array of `Rectangle` and `Circle` objects.

Problem Analysis: We can sort objects if we can compare them. In fact, objects that are comparable are easy to sort, but different objects compare differently with each other. The above software engineering tip therefore suggests proceeding in three steps:

Step 1: We need a type that provides a comparison property. It is not important to specify *how* to compare two entities, just that they *can* be compared. Therefore we create an interface `Sortable` to provide a `lessThan` method that decides if an object is less than the input object.

Step 2: Since objects of type `Sortable` can be compared with each other we create a `Sorter` class that implements the following sorting algorithm:

- find the smallest element in the array and swap it with the first element of the array
- find the smallest element in the array except for the first one and swap it with the second element of the array
- find the smallest element in the array except for the first two and swap it with the third element of the array
- continue in this fashion until the array is sorted

Step 3: We want to sort rectangles and circles so we need to ensure that these classes implement `Sortable`. Therefore we must define the `lessThan` method using, for example, the `area` to compare `Rectangle` and `Circle` objects. Then we can use the `Sorter` class to sort arrays of `Rectangle` and `Circle` objects.

Project Implementation: We need an interface `Sortable`, a class `Sorter`, and classes with specific implementations of the comparison property of the `Sortable` interface.

Step 1: The interface `Sortable` needs a method to compare `Sortable` objects so we include a single `abstract` method:

```
public interface Sortable
{  public abstract boolean lessThan(Sortable obj); }
```

That interface defines a new type, which can be used by `Sorter`.

Step 2: The `Sorter` class needs a method to sort. That method takes as input an array of `Sortable` objects, because they are comparable via `lessThan`. According to the above algorithm we need to find

the smallest element in parts of an array and swap it with another so we create utility methods `findMin` and `swap`, respectively. All methods depend only on input parameters so they can be `static`.

```
public class Sorter
{  public static void sort(Sortable[] A)
   {   for (int i = 0; i < A.length-1; i++)
         swap(A, i, findMin(A, i));
   }
   private static int findMin(Sortable[] A, int start)
   {  // returns index of smallest element in A[start], A[start+1], ...
      int min = start;
      for (int i = start+1; i < A.length; i++)
         if (A[i].lessThan(A[min]))
            min = i;
      return min;
   }
   private static void swap(Sortable[] A, int pos1, int pos2)
   {  // swaps A[pos1] with A[pos2]
      Sortable temp = A[pos1];
      A[pos1] = A[pos2];
      A[pos2] = temp;
   }
}
```

The `Sorter` class compiles without problems as long as the `Sortable` interface is saved into the same directory but we cannot test it because `Sortable` is an interface and `lessThan` is abstract.

Step 3: Now we define `Rectangle` and `Circle` so that they are of type `Sortable` *in addition* to any other type. `Rectangle` and `Circle` (see examples 3.23 and 3.32) extend `Shape`. If we redefine `Shape` to implement `Sortable`, every class extending `Shape` is also of type `Sortable`. To implement `Sortable` we must define the method `lessThan` to compare shapes according to their area.

```
public class Shape implements Sortable
{  protected String name;
   protected double area, perimeter;

   public String toString()
   {  return name +" [area = " +area +", perimeter = " +perimeter +"]"; }
   public void display()
   {  System.out.println(toString()); }
   public boolean lessThan(Sortable obj)
   {  if (obj instanceof Shape)
         return (area < ((Shape)obj).area);
      else
         return false;
   }
}
```

All classes extending `Shape` are also of type `Sortable` and hence can be sorted by `Sorter`.

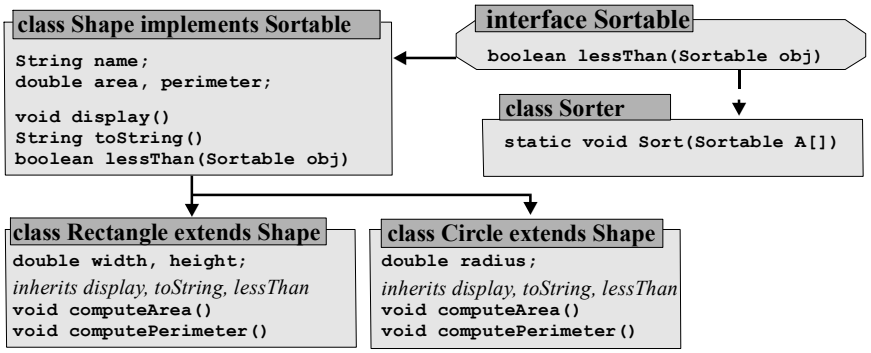


Figure 3.49: The `Shape` *hierarchy with associated* `Sortable` *interface and* `Sorter` *class*

Here is a test program to create and sort arrays of `Rectangle` and `Circle` objects (see figure 3.50):

```
public class SortingShapes
{  public static void showShapes(String title, Shape shapes[])
   {  System.out.println(title);
      for (int i = 0; i < shapes.length; i++)
         System.out.println(shapes[i]);
   }
   public static void main(String args[])
   {  Rectangle r[]={new Rectangle(2,6),new Rectangle(3,3),new Rectangle(1,4)};
      Circle    c[]={new Circle(3),     new Circle(2),     new Circle(1)};
      for (int i = 0; i < r.length; i++)
      {  r[i].computeArea();
         r[i].computePerimeter();
         c[i].computeArea();
         c[i].computePerimeter();
      }
      showShapes("Unsorted Rectangles", r);
      Sorter.sort(r);
      showShapes("Sorted Rectangles", r);
      showShapes("Unsorted Circles", c);
      Sorter.sort(c);
      showShapes("Sorted Circles", c);
   }
}
```

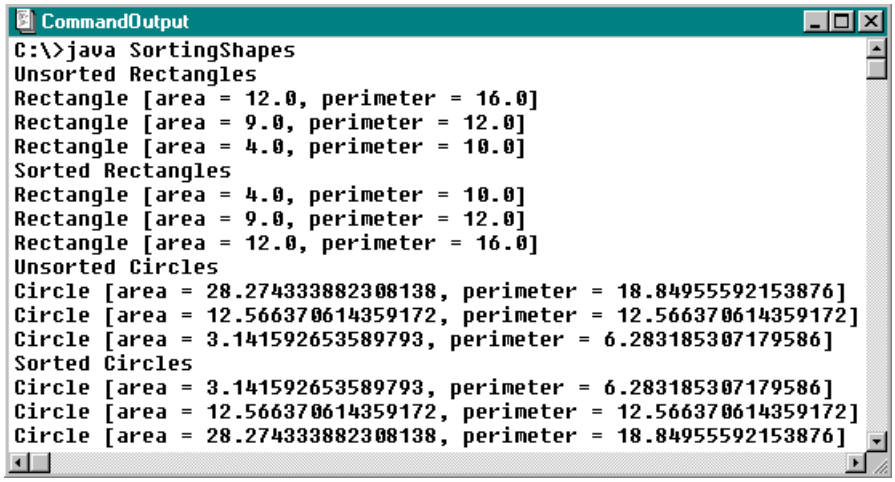


Figure 3.50: Output of `SortingShapes` *class*

■

We have accomplished a lot more than may be apparent at first because our `Sorter` class can sort objects of *any* type with little effort. All we have to do is to change a class so that it implements `Sortable` and defines `lessThan`. An array of objects of that class is a valid input variable for `Sorter.sort`, which can sort the array without further code adjustments.

We could even sort an array of `Shapes` such as

```
Shape s[] = {new Rectangle(2.0, 3.0), new Circle(4.0),
             new Rectangle(1.0, 2.0), new Circle(2.0),
             new Rectangle(3.0, 0.5), new Circle(3.0)};
```

It is left as an exercise to determine whether the rectangles will appear first or last in the array after it has been sorted. [37]

Our `Sorter` class does require existing classes to change before objects can be sorted. That makes it difficult to provide variable comparison methods. For example, how could you sort an array of objects once in increasing order and the second time in decreasing order? In example 3.40 we will revisit this problem, but first we provide at another example to illustrate the usefulness of interfaces.

Example 3.39: A general-purpose menu system using classes and interfaces

Many simple programs use a text-based menu, where the user types a letter to cause one of several possible actions.[38] Create a generally useful mechanism that all programs that which to use text-based menus can utilize. Use the `Console` class from section 2.4 to obtain user input.

Problem Analysis: Programs using a text-based menu usually work as follows:

- a menu of choices is displayed on the screen
- the user is asked to choose an option by typing one or more keys
- depending on the user's choice, certain actions take place
- the menu is displayed again until the user chooses some exit key combination

These steps form an algorithm that would work if we had objects that could perform an action, but the action performed should be different for different programs. Again our software engineering tip applies so we proceed in three steps:

1. We create an interface `MenuUser` that provides an abstract method `performAction`
2. We create a `Menu` class that displays choices, asks for user input, and calls on `performAction` with the user's selection as input to perform some unspecified action.
3. A class that wants to use the interface must implement `MenuUser` and provide a concrete implementation of `performAction`. It can then use `Menu` to interact with the user.

Project Implementation: We implement an interface `MenuUser` and a class `Menu` to handle the user interaction and decide when to call `performAction`. A third class `MenuTest` tests whether our mechanism works correctly.

Step 1: The interface contains a single `abstract` method.

[37] You need to add the methods `public abstract void computeArea()` and `public abstract void computePerimeter()` to `Shape` and declare the class as `abstract` so that polymorphism applies (see section 3.6).
[38] Nowadays text-based menus are no longer in vogue and graphical user interfaces as introduced in chapter 4 are preferred. But text-based menu programs are still useful to create quick solutions for simple tasks.

```
public interface MenuUser
{  public abstract void performAction(String command); }
```

Step 2: The `Menu` class displays a menu and calls a `MenuUser`'s `performAction` method when an action is necessary. It contains a field `choices` of type array of `String` to represent the menu options, a field `title` for the title of the menu, and a field `user` of type `MenuUser` to provide a `performAction` method. The fields are initialized in the constructor, which then calls a utility method named `start`.

```
public class Menu
{  private String title;
   private String choices[] = null;
   private MenuUser user = null;

   public Menu(String _title, String _choices[], MenuUser _user)
   {  title = _title;
      choices = _choices;
      user = _user;
      start();
   }
   private String getMenuChoice()
   {  System.out.println("\n\t" + title + "\n");
      for (int i = 0; i < choices.length; i++)
         System.out.println("\t" + choices[i]);
      System.out.println("\n\tType 'Exit' to exit the menu");
      System.out.print("\n\tEnter your choice: ");
      return Console.readString();
   }
   private void start()
   {  String choice = getMenuChoice();
      while (!choice.equalsIgnoreCase("exit"))
      {  user.performAction(choice);
         choice = getMenuChoice();
      }
   }
}
```

The `start` method display the menu choices, waits for user input, and calls `performAction` until the string `"exit"` is entered. Since `performAction` as provided by the field `user` of type `MenuUser` is `abstract`, no particular action occurs.

Step 3: `MenuUser` and `Menu` will compile but we need a third class providing a concrete version of `performAction` to test our system.

```
public class MenuTest implements MenuUser
{  private final String[] MAIN_MENU = {"[A] compute 10*20",
                                       "[T] for trig menu ..."};
   private final String[] SUB_MENU  = {"[1] to compute sin(2)",
                                       "[2] to compute cos(2)"};
   private Menu mainMenu = null;
   private Menu subMenu = null;

   public MenuTest()
   {  mainMenu = new Menu("Multiplication", MAIN_MENU, this); }
   public void performAction(String command)
   {  if (command.equalsIgnoreCase("A"))
         System.out.println("Result: " + (10*20));
      else if (command.equalsIgnoreCase("T"))
         subMenu = new Menu("Trigonometry", SUB_MENU, this);
      else if (command.equalsIgnoreCase("1"))
         System.out.println("sin(2) = " + Math.sin(2.0));
```

```
        else if (command.equalsIgnoreCase("2"))
           System.out.println("cos(2) = " + Math.cos(2.0));
     }
     public static void main(String args[])
     {  MenuTest mt = new MenuTest(); }
}
```

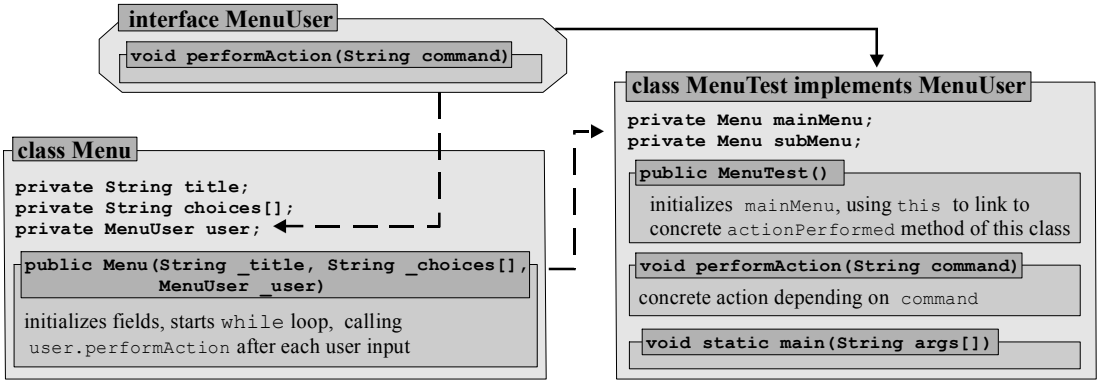
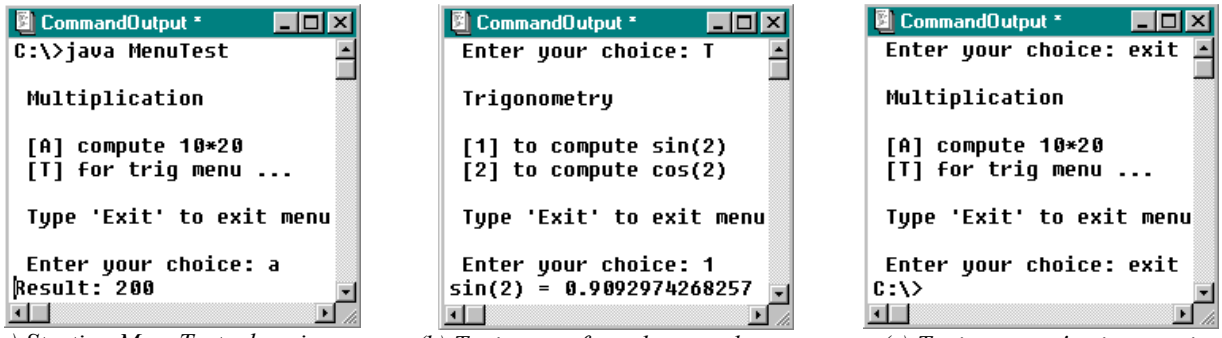


Figure 3.51: The relationships between `MenuUser`, `Menu`, *and* `MenuTest`

`MenuTest` uses two fields of type `Menu` named `mainMenu` and `subMenu`. It implements `MenuUser` so it must define `performAction`. The constructor of `MenuTest` instantiates the first `Menu` object, using `this` as input matched to the `user` parameter of `Menu`[39]. That compiles because `MenuTest` implements `MenuUser` so that `this` is of type `MenuUser`. When the `Menu` class executes the `user`'s `performAction` method, the concrete version provided by `MenuTest` is used because `MenuTest` has been initialized as `user`. When option `'T'` is chosen, another version of `Menu` is instantiated, representing the submenu. The second version of `Menu` will enter its `while` loop, again calling the `user`'s `performAction` method when appropriate. If the user types `'exit'`, the loop finishes so that the second instance of `Menu` is done. The first instance is still executing *its* `while` loop so that the original menu reappears. If `'exit'` is typed again, this instance of `Menu` also finishes, which means that the constructor of `MenuTest` can exit and the program closes. Figure 3.52 shows that everything works as it should.



(a) Starting MenuTest, choosing `'a'` *(b) Typing* `'T'` *for submenu, then* `'1'` *(c) Typing* `'exit'` *twice to exit*

Figure 3.52: Testing the `Menu`, `MenuUser`, *and* `MenuTest` *system*

■

`MenuUser` and `Menu` can be used in every program that needs a text-based menu. They handle the interaction with the user and can be adapted to a particular program by a concrete `performAction` method. The exercises will provide additional details.

[39] We can not initialize `mainMenu` in the field section with `this` as input, because `this` refers to the current object, which only comes into existence when the constructor is called.

Example 3.40: Sorting arrays using a `FlexSorter` *class and* `Comparable` *interface*

The `Sorter` class from example 3.38 can sort arrays of `Sortable` objects. That means that existing classes must change to implement `Sortable` before they can be sorted. Create a more flexible class to sort arrays of objects without changing the class of the objects. Test it to sort an array of `Length` objects (see example 3.12 for a definition of `Length`)

Problem Analysis: We still need to compare objects before we can sort them. In our previous approach we created a `Sortable` interface and objects that wanted to be sorted had to implement `Sortable` and define a `lessThan` method. This time we want to use a method external to existing classes to compare object so that the objects themselves do not have to provide a comparison method. The new method must decide how to compare two objects and is not supposed to be part of the objects. It must therefore take two objects as input.

Project Implementation: We create an interface that we call `Comparable`. It contains a method `compare` that decides how two objects compare:

```
public interface Comparable
{  /** The compare method should return:
          -1 if obj1 < obj2, 0 if obj1 = obj2, +1 if obj1 > obj2 */
   public abstract int compare(Object obj1, Object obj2);
}
```

The conceptual difference to the `Sortable` interface is that `compare` uses *two* objects as input while `lessThan` compares an existing object to an input object.

We must change the `Sorter` class so that it takes as input an array of `Object`, not `Sortable`. To sort, it uses the `Comparable` type, providing the `compare` method to compare objects. We call the new sorting class `FlexSort`.[40] The changes to the previous code are minimal and are shown in italics.

```
public class FlexSorter
{  public static void sort(Object[] A, Comparable comp)
   {  for (int i = 0; i < A.length; i++)
         swap(A, i, findMin(A, i, comp));
   }
   private static int findMin(Object[] A, int start, Comparable comp)
   {  // returns index of smallest element in A[start], A[start+1], ...
      int min = start;
      for (int i = start+1; i < A.length; i++)
         if (comp.compare(A[i], A[min]) < 0)
            min = i;
      return min;
   }
   private static void swap(Object[] A, int pos1, int pos2)
   {  // swaps A[pos1] with A[pos2]
      Object temp = A[pos1];
      A[pos1] = A[pos2];
      A[pos2] = temp;
   }
}
```

This class sorts an array of `Object` but requires an instance of `Comparable` to provide a particular `compare` method. To sort an array of `Length` objects we must first create a class that implements

[40] Instead of creating a new class we could have overloaded the methods in the `Sorter` class, but then the `Sorter` class requires the interface `Comparable` *and* `Sortable` to be present before it compiles.

`Comparable` to decide how two `Length` objects should compare. Recall that a `Length` has a `scale` and a `value` and provides methods to handle conversion from one scale to another. In example 3.12 we defined a `Length` class to handle lengths in feet and meter, including set/get methods. We decide to sort `Length` objects by comparing their values in meter.

```
public class LengthComparer implements Comparable
{  public int compare(Object obj1, Object obj2)
   {  if ((obj1 instanceof Length) && (obj2 instanceof Length))
      {  Length length1 = ((Length)obj1).convertToMeter();
         Length length2 = ((Length)obj2).convertToMeter();
         if (length1.getValue() < length2.getValue())
            return -1;
         else if (length1.getValue() == length2.getValue())
            return 0;
         else
            return 1;
      }
      else
         return -1;
   }
}
```

The following program will test our new `FlexSorter` class, sorting an array of `Length` objects. Its output is shown in figure 3.53:

```
public class FlexSorterTest
{  public static void main(String args[])
   {  Length lengths[] = { new Length(3, "meter"), new Length(3, "feet"),
                           new Length(1, "meter"), new Length(1, "feet")};
      System.out.println("Unsorted lengths:");
      for (int i = 0; i < lengths.length; i++)
         lengths[i].showLength();
      FlexSorter.sort(lengths, new LengthComparer());
      System.out.println("Sorted lengths");
      for (int i = 0; i < lengths.length; i++)
         lengths[i].showLength();
   }
}
```

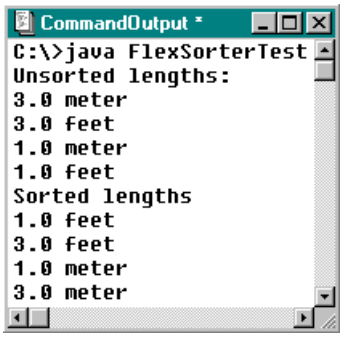


Figure 3.53: Output of `FlexSorterTest`

■

`FlexSorter` is our third generally useful class,[41] in addition to `List` (section 3.5) and `Console` (section 2.4). We should provide full documentation using the `javadoc` tool, which is left as an exercise.

[41] In the *Case Study* section we change `FlexSorter` so that it can sort an entire array as well as parts of an array. The changes are minimal but make our class even more flexible.

It may be confusing to create your own interfaces, but it is easy to use existing ones. Java includes many predefined interfaces, which we use in chapter 4 to deal with events generated by programs with a graphical user interface.

Polymorphism

Polymorphism is the last concept in object-oriented programming we need to explore. It simplifies using related classes with common features so that programs can easily 'grow' to include new classes with additional functionality.

Polymorphism

Polymorphism is the ability to automatically select appropriate methods and objects from an inheritance hierarchy. For polymorphism to work the hierarchy of classes must have at least one common feature such as a shared method header.

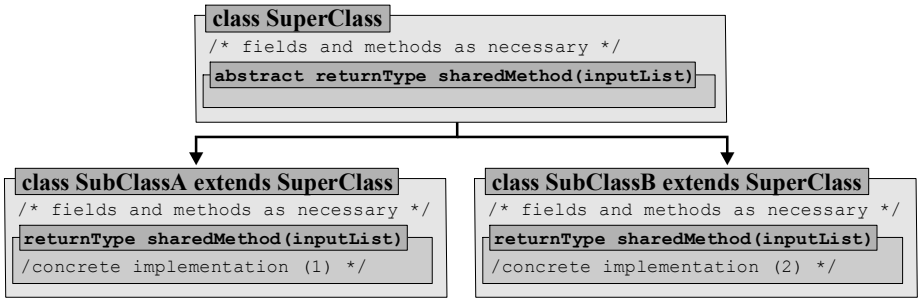


Figure 3.54: Inheritance hierarchy where polymorphism can select which `sharedMethod` *executes*

Example 3.41: Illustrating polymorphism

Define a superclass containing an `abstract` method and two subclasses that provide different implementations of that method. Create a test class with an array of elements of the superclass type that contains one object from each subclass. Use a loop to call the shared method of each object. Which version of the method is called?

To create the class hierarchy is easy:

```
public abstract class SuperClass
{  public abstract void display(); }

public class SubClassA extends SuperClass
{  public void display()
   {  System.out.println("This is subclass A"); }
}

public class SubClassB extends SuperClass
{  public void display()
   {  System.out.println("This is subclass B"); }
}
```

The test class needs a standard `main` method to be executable. It declares an array of type `SuperClass`, but since `SuperClass` is abstract in can not instantiate objects. We add an object of type

SubClassA and another of type SubClassB to the array, which works because both objects are also of type SuperClass. Then we use a for loop to call on display for each object in the array.

```
public class PolyTester
{  public static void main(String args[])
   {  SuperClass array[] = new SuperClass[2];
      array[0] = new SubClassA();
      array[1] = new SubClassB();
      for (int i = 0; i < array.length; i++)
         array[i].display();
   }
}
```

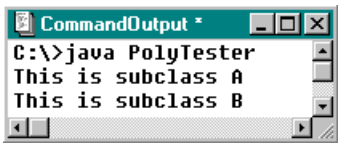


Figure 3.55: Executing PolyTester

The output of PolyTester is shown in figure 3.55. When the display method of object array[0] is called, the JVM knows that the object is of type SubClassA in addition to SuperClass and chooses to execute the display method of SubClassA. When executing array[1].display, the object is now of type SubClassB and the JVM calls *its* version of display. Polymorphism allows the JVM to decide, at the time the program executes, which version of the common display method to use based on the true type of the current object.

■

This example may not look striking, but the same concept can be applied to considerably simplify adding new classes and functionality to existing programs as long as the participating classes were designed with polymorphism in mind. The prime example of the usefulness of polymorphism is the toString method shared by all objects.

Example 3.42: Using polymorphism to call toString

Create a class Address, Rectangle, Length, and Check. Choose some implementation but make sure that each class provides its own implementation of the toString method inherited from Object. Create an array of these objects and display each one.

We have seen these classes in earlier examples. Here are simple versions of these classes:

```
public class Address
{  private String name, email;
   public Address(String _name,
                  String _email)
   {  name = _name;
      email = _email;
   }
   public String toString()
   {  return name + " (" + email + ")"; }
}
public class Length
{  private String scale;
   private double value;
   public Length(double _value,
                 String _scale)
   {  value = _value;
      scale = _scale;
   }
   public String toString()
   {  return value + " " + scale; }
}
```

```
public class Rectangle
{  private double width, height;
   public Rectangle(double _width,
                    double _height)
   {  width = _width;
      height = _height;
   }
   public String toString()
   {  return width + " x " + height; }
}
public class Check
{  private String descr;
   private double amount;
   public Check(String _descr,
                double _amount)
   {  desrc = _descr;
      amount= _amount;
   }
   public String toString()
   {  return descr + " $" + amount; }
}
```

We can use one array of type Object to store these types because every class extends Object.

```
public class TestAutoString
{  public static void main(String args[])
   {  Object array[] = new Object[4];
      array[0] = new Address("Bert Wachsmuth", "wachsmut@shu.edu");
      array[1] = new Check("Regular salary", 6500);
      array[2] = new Length(5.11, "feet");
      array[3] = new Rectangle(3, 4);
      for (int i = 0; i < array.length; i++)
         System.out.println(array[i]);
   }
}
```

Since `toString` is part of every class, `System.out.println` calls it automatically, but the particular version of `toString` is chosen by polymorphism as the program executes. If a class overrides `toString` to provide useful information about itself, that version is used, otherwise the default version of `Object` is substituted. The output is shown in figure 3.56.

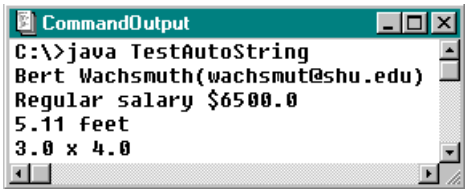


Figure 3.56: Output of `TestAutoString`

■

Software Engineering Tip: You should consider polymorphism whenever you create a hierarchy where all classes can share a common method header but each one needs a different implementation of that method. Use a generic or abstract method in the superclass and override or implement it in the subclasses to enable polymorphism.

If multiple overridden methods are available for a class, polymorphism picks the one that is closest to the true type of an object, eliminating the need for typecasting.

Example 3.43: Using polymorphism for `Shape` *hierarchy*

Redesign the classes `Shape`, `Rectangle`, `Circle`, and `Square` from examples 3.23 and 3.25 as necessary so that a program using them can utilize polymorphism to compute their area and perimeter and to display them.

Problem Analysis: Each geometric shape has an area and perimeter but computes them differently. To enable polymorphism we need to make sure that all shapes can compute their area and perimeter, but provide different implementations for the computations. Therefore the superclass `Shape` contains the `abstract` methods `computeArea` and `computePerimeter`, which are implemented and overridden in the various subclasses, if necessary. We also add a `toString` method to `Shape` that displays the basic properties of shapes. Here are our classes:

```
public abstract class Shape
{  protected String type;
   protected double area, perim;

   public abstract void computeArea();
   public abstract void computePerimeter();
   public String toString()
   {  return type + ":\n\tArea = " + area +
                    "\n\tPerimeter = " + perim;
   }
}
```

```
public class Rectangle extends Shape
{  protected double width, height;
   public Rectangle(double _width,
                    double _height)
   {  type = "Rectangle";
      width = _width;
      height = _height;
   }
   public void computeArea()
   {  area = width * height; }
   public void computePerimeter()
   {  perim = 2*width + 2*height; }
   public String toString()
   {  return super.toString() +
             "\n\tWidth = " + width +
             "\n\tHeight = " + height;
   }
}
```

```
public class Square extends Rectangle
{  public Square(double _side)
   {  super(_side, _side);
      type = "Square";
   }
}
```

```
public class Circle extends Shape
{  protected double radius;
   public Circle(double _radius)
   {  type = "Circle";
      radius = _radius;
   }
   public void computeArea()
   {  area = Math.PI * radius * radius; }
   public void computePerimeter()
   {  perim = 2.0 * Math.PI * radius; }
   public String toString()
   {  return super.toString() +
             "\n\tRadius = " + radius;
   }
}
```

Now we can easily create instances of these types and let polymorphism pick the most appropriate methods to deal with them.

```
public class ShapeTestWithPolymorphism
{  public static void main(String args[])
   {  Shape shapes[] = { new Rectangle(2, 4), new Circle(3),
                         new Square(4)};
      for (int i = 0; i < shapes.length; i++)
      {  shapes[i].computeArea();
         shapes[i].computePerimeter();
         System.out.println(shapes[i]);
      }
   }
}
```

The output of executing S`hapeTestWithPolymorphism` is shown in figure 3.57. Note in particular that for `shapes[2]`, which is of type `Square`, `Rectangle`, and `Shape` (and `Object`), the `toString` method of the `Rectangle` was picked, which is the type closest to `Square`.

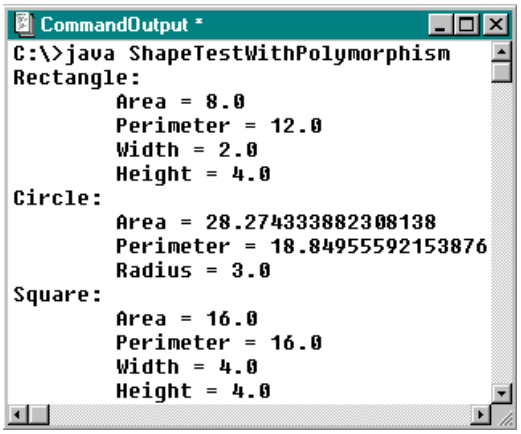


Figure 3.57: Output of `ShapeTestWithPolymorphism`

■

When we add a new class to our `Shape` collection, polymorphism will allow us to do so with little modifications.

Example 3.44: Adding `RightTriangle` *to* `Shape` *hierarchy*

Add a `RightTriangle` to the collection of `Shapes` and modify the `ShapeTester` program accordingly.

We consider a `RightTriangle` as a special case of a `Rectangle`:

```
class RightTriangle extends Rectangle
{  RightTriangle(double _width, double _height)
   {  super(_width, _height);
      type = "Triangle";
   }
   public void computeArea()
   {  area = 0.5 * width * height; }
   public void computePerimeter()
   {  perim = width + height + Math.sqrt(width*width+height*height); }
}
```

The advantage of the way we constructed our various `Shape` classes becomes apparent when we add a new triangle to `ShapeTestWithPolymorphism`. All we have to do is to add a new triangle object to the array of shapes and the rest is handled automatically.[42]

```
public class ShapeTestWithPolymorphism
{  public static void main(String args[])
   {  Shape shapes[] = { new Rectangle(2, 4), new Circle(3),
                         new Square(4), new RightTriangle(3, 4)};
      for (int i = 0; i < shapes.length; i++)
      {  shapes[i].computeArea();
         shapes[i].computePerimeter();
         System.out.println(shapes[i]);
      }
   }
}
```

[42] Inheritance is not perfect. The field names `width` and `height` and the `toString` method of `RightTriangle` are inherited from `Rectangle`, but the fields should really be called `base` and `height`. That can only be accomplished if `RightTriangle` extends `Shape` instead of `Rectangle`, but then we must implement its own `toString` method.

■

Case Study: OO Design for Invoice Program

We have explored all concepts of object-oriented programming. In this section, which is optional, we create a program using everything we have learned so far.

Example 3.45: The Acme Widget Company Invoicing System

The owner of the Acme Widget Company (AWC) wants to hire you to create a program to manage her inventory and create invoices for items ordered. Take the job.

This problem is too vague to be solved right away, which is typical for real-word programming tasks.

Software Engineering Tip: The following guidelines help to apply object-oriented programming techniques to real-world programming jobs:

(1) Understand: Write down exactly what the problem entails and what the deadlines are. Identify tasks you cannot do. Discuss your understanding with the customer and other users.
(2) Analyze: Identify entities as models for classes. Use "has-a", "does-a", and "is-a" to identify fields, methods, and inheritance. Collect useful existing classes and ask other programmers.
(3) User Interface: Sketch a rough layout for a user interface. Look at comparable programs. Discuss your layout with the customer and potential users.
(4) Generalize: Look for general concepts and identify classes that can serve as superclasses. Identify classes that can be useful for other projects.
(5) Divide and Conquer: Divide problem into categories (classes) and categories into tasks (methods). Create class skeletons and specify field types, method headers, and hierarchies.
(6) Implement: Implement classes designed in (5). Modify class skeletons if necessary. Make classes as flexible as possible. Restrict user interaction to as few classes as possible.
(7) Test: Test classes individually. Test program to solve original problem. Take user errors and extreme cases into account. Let customer test your program. Refine classes and retest.
(8) Document: Provide complete documentation and supporting documents for your program. Clarify usage instructions and limitations of your program.

Step 1: Understand: After extensive discussions with the owner and employees of AWC we understand that our program should do the following:

- *Manage an inventory list.* Our program should manage an inventory of labeled items with descriptions, price, and available units. Employees need to add and remove units, but the item labels, descriptions, and prices are not likely to change. Items are sorted by label.
- *Create invoices.* We need to create invoices for customers ordering items from the inventory. Invoices consist of customer name, a sorted list of items ordered, and the total price.
- The company already has a database of customers that should be interfaced with the invoicing program.

Based on our knowledge of Java we know we cannot save data to file, we cannot print, we can only create text-based programs, and we cannot interface Java programs with other programs. We offer to create a program with these limitations and discuss deadlines and costs. The owner of AWC agrees.

Step 2: Analyze: Based on our understanding there are several distinct entities: inventory items, an inventory, invoice items, and an invoice.

- The *inventory* consists of a sorted *list of inventory items*. Each item has a *label*, *description*, *price*, and number of *units*. We need to *add* and *remove units* for each inventory item.
- An *invoice* contains a sorted *list of items* from the inventory, number of *units* ordered, their *subtotal*, a customer *name* and a *total* price. We need to be able to *add* and *delete* items.

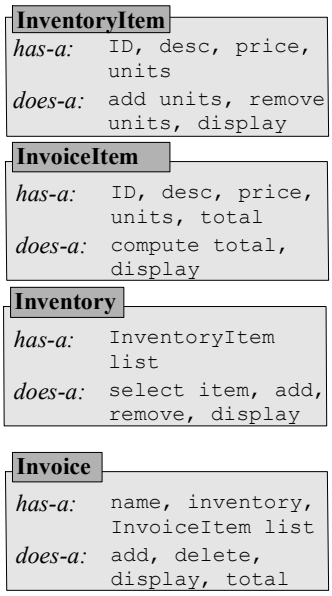
We identify the following entities as possible classes:

`InventoryItem`:
- *has-a*: label, description, price, available units
- *does-a*: increase and decrease available units, display

`InvoiceItem`:
- *has-a*: label, description, price, units ordered, subtotal
- *does-a*: compute subtotal, display

`Inventory`:
- *has-a*: sorted list of `InventoryItems`
- *does-a*: select `InvoiceItem`, add or remove units to `InvoiceItem`, display list

`Invoice`:
- *has-a*: name, total price, sorted list of `InvoiceItems`
- *does-a*: add and delete `InvoiceItem`, compute total, display list
- *needs to know the* `Inventory`



`Invoice` and `Inventory` need to store sorted lists of items, so `List` (example 3.34) together with `FlexSorter` (example 3.40) may be useful. To interact with the user we could use `Menu` and `MenuUser` (example 3.39). To obtain user input we use the `Console` class (section 2.4).

Step 3: User Interface: We suggest the following text-based menu structure:

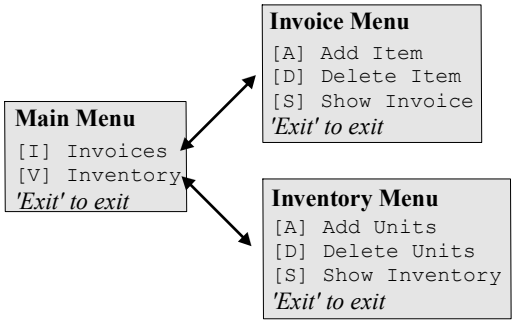


Figure 3.58: Suggested menu-based user-interface

Step 4: Generalize: We need to identify possible superclasses and generally useful classes:

- `InventoryItem` and `InvoiceItem` contain a number of common fields so we create a superclass `Item` to factor out common fields and tasks.

- `Inventory` and `Invoice` contain sorted lists of objects that the user can manipulate. Instead of using a field we could implement these classes as subclasses of `List`. Implementing `MenuUser` can provide user interaction.
- To interact with the user we need to prompt the user to enter a value and read the response using the `Console` class. To simplify user input we create a `Util` class with methods:
  - `String getString(String prompt)`: Displays prompt, returns user input as `String`
  - `int getInt(String prompt)`: Displays prompt, returns user input as `int`
- We identified `List` as a possible superclass, but it does not provide searching and sorting. Instead of searching and sorting `Inventory` and `Invoices` directly we create a `SortedList` class extending `List` that keeps items sorted and provides a general search mechanisms.

Step 5: Divide and Conquer: We have so far identified the existing classes `List`, `FlexSorter`, `Comparable`, `Menu`, and `MenuUser`, which should be collected into one folder and reviewed. Figure 3.59 show the class skeletons for the `Item` hierarchy of classes, including field types and method headers.

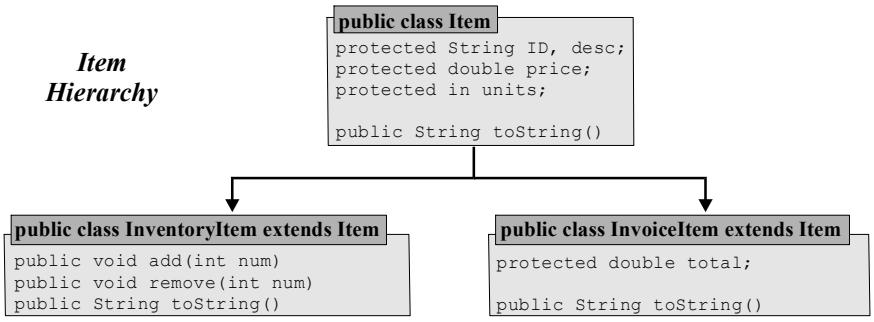


Figure 3.59: The hierarchy of `Item` *classes*

The design for `SortedList` and its subclasses `Inventory` and `Invoice` are shown in figure 3.60:

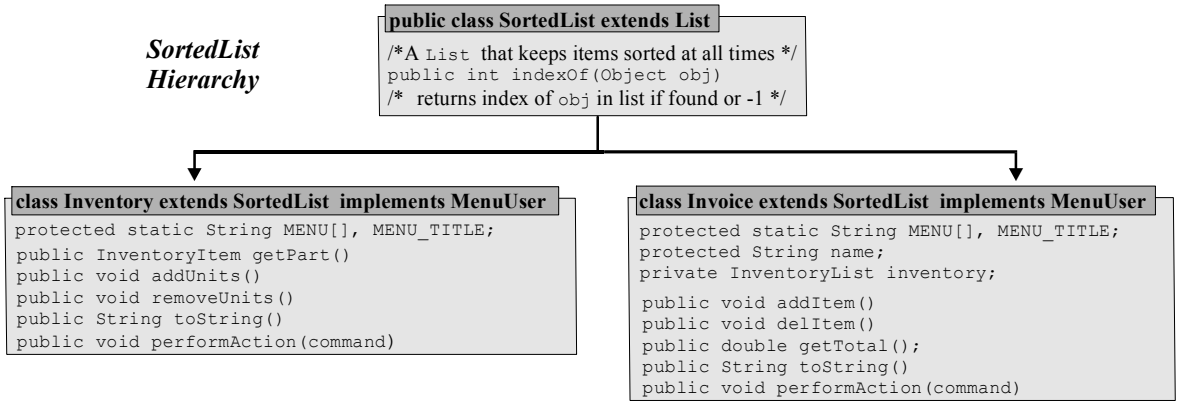


Figure 3.60: Outline of `SortedList` *class and its subclasses* `Inventory` *and* `Invoice`.

We add constructors and a master class with a `main` method during the implementation stage.

Step 6: Implement: We start by implementing the generally useful classes. The `Util` class contains methods `getString` and `getInt` to prompt the user to enter a value and return a `String` or an `int`. We add a method `dollar` to convert a `double` to a `String` representing currency (see section 1.5).

```
import java.text.*;

public class Util
{  private static DecimalFormat formatTool = new DecimalFormat("#,###.00");
```

```
   public static String dollar(double num)
   {  return "$" + formatTool.format(num); }
   public static String getString(String prompt)
   {  System.out.print(prompt + ": ");
      return Console.readString().trim();
   }
   public static int getInt(String prompt)
   {  System.out.print(prompt + ": ");
      return Console.readInt();
   }
}
```

To create `SortedList` we need to extend `List` and ensure that items are always sorted. We could use the `FlexSorter` class to sort the items after every addition, but we need access to the `private` field `list` of the `List` class and we need to sort only part of that array. Therefore we modify List to declare its field list as protected, not private, and we overload the `sort` method of the `FlexSorter` class so that it can also sort parts of an array.[43]

```
public class List
{  protected Object[] list = null;
   /* everything else remains unchanged */
}

public class FlexSorter
{  public static void sort(Object[] A, Comparable comp)
   {  sort(A, A.length, comp); }
   public static void sort(Object[] A, int stop, Comparable comp)
   {  for (int i = 0; i < stop; i++)
         swap(A, i, findMin(A, i, stop, comp));
   }
   private static int findMin(Object[] A, int start, int stop, Comparable comp)
   {  // returns index of smallest element in A[start],A[start+1],...,A[stop-1]
      int min = start;
      for (int i = start+1; i < stop; i++)
         if (comp.compare(A[i], A[min]) < 0)
            min = i;
      return min;
   }
   private static void swap(Object[] A, int pos1, int pos2)
   { /* as before, no change */ }
}
```

Now `SortedList` can extend `List` and override the `add` method to sort the array `list` after every addition.[44] We also add a method `public int indexOf(Object obj)`, which compares `obj` with every element in the list. If it finds a match as determined by `Comparable.compare` it returns its index, otherwise –1. An instance of `Comparable` is initialized in the constructor.

```
public class SortedList extends List
{  private Comparable comp;

   public SortedList(Comparable _comp)
   {  comp = _comp; }
   public SortedList(int _maxItems, Comparable _comp)
```

[43] That modification does not change the previous functionality of `List` and `FlexSorter`, so it does not break the `public` contract of these classes.

[44] We can use `FlexSorter` to sort but it makes the `add` method inefficient. This is a common problem with object-oriented programming: existing objects offers easy solutions, but to create optimal solutions new code needs to be written. The benefit is that we *know* that `FlexSorter` works, whereas new code may introduce new bugs.

```
      {  super(_maxItems);
         comp = _comp;
      }
      public void add(Object obj)
      {  super.add(obj);
         FlexSorter.sort(list, getSize(), comp);
      }
      public int indexOf(Object obj)
      {  for (int i = 0; i < getSize(); i++)
            if (comp.compare(get(i), obj) == 0)
               return i;
         return -1;
      }
   }
```

After taking care of our generally useful classes, we need to create the classes specific to our project. The `Item` class and its subclass do not need to deal with user input and are defined as follows:

```
   public class Item
   {  protected String ID, desc;
      protected double price;
      protected int    units;

      public Item(String _ID)
      {  ID = _ID; }
      public String toString()
      {  return ID + "\t" + units + " units x " + Util.dollar(price) + "\t"; }
   }
```

`InventoryItem` and `InvoiceItem` extend `Item`. An `InventoryItem` initializes its values through its constructor. An `InvoiceItem` is always generated from an `InventoryItem` so its constructor uses an `InventoryItem` as input and copies the relevant values into its own fields.

```
public class InventoryItem extends Item
{  public InventoryItem(String _ID,
                        String _desc,
                        double _price,
                        int _units)
   {  super(_ID);
      desc = _desc;
      price = _price;
      units = _units;
   }
   public void remove(int num)
   {  units -= num; }
   public void add(int num)
   {  units += num; }
   public String toString()
   {  return super.toString() + "\t"
             + desc;
   }
}
```

```
public class InvoiceItem extends Item
{  protected double total;

   public InvoiceItem(InventoryItem item,
                      int _units)
   {  super(item.ID);
      desc = item.desc;
      price = item.price;
      units = _units;
      total = price * units;
   }
   public String toString()
   {  return super.toString() +" = " +
             Util.dollar(total) +
             "\t(" + desc + ")";
   }
}
```

To compare objects of these types we create a class implementing `Comparable`, which can be used as input to the `SortedList` constructor. We consider items to be equal if their `ID` (label) matches.

```
   public class ItemComparer implements Comparable
   {  public int compare(Object obj1, Object obj2)
      {  if ( (obj1 instanceof Item) && (obj2 instanceof Item))
         {  Item item1 = (Item)obj1;
```

```
            Item item2 = (Item)obj2;
            return item1.ID.compareTo(item2.ID);
         }
         else
            return -1;
      }
   }
```

For example, if `A = new Item("0001") and B = new InventoryItem("0001","Junk",1.23,5)` then `itemComparer.compare(A, B)` returns `0`, i.e. `A` and `B` are considered the same.

The `Inventory` class extends `SortedList` and implements `MenuUser` so it needs an appropriate constructor and a `performAction` method. To create a basic inventory, the constructor adds a few sample `InventoryItems` (which should be sorted automatically). If user input is necessary, the methods `getString` and `getInt` from the `Util` class are used.

```
public class Inventory extends SortedList implements MenuUser
{  protected final static String MENU[] =
        {"[A] Add Units", "[D] Delete Units", "[S] Show Inventory"};
   protected static final String TITLE = "Inventory Menu";

   public Inventory()
   {  super(4, new ItemComparer());
      add(new InventoryItem("0001", "Basic Widget", 3.44, 15));
      add(new InventoryItem("3002", "Expensive Widget", 1203.00, 10));
      add(new InventoryItem("0011", "Sales Widget", 123.00, 10));
      add(new InventoryItem("2001", "Safety Widget", 33.99, 20));
   }
   public InventoryItem getPart()
   {  System.out.println(this);
      int index = indexOf(new Item(Util.getString("Enter ID of item")));
      if (index >= 0)
         return (InventoryItem)get(index);
      else
         return null;
   }
   public void addUnits()
   {  InventoryItem item = getPart();
      if (item != null)
         item.add(Util.getInt("Number of units to add"));
      else
         System.out.println("Inventory item not found");
   }
   public void removeUnits()
   {  InventoryItem item = getPart();
      int units = Util.getInt("Number of units to remove");
      if ((item != null) && (item.units >= units) && (units >= 0))
         item.remove(units);
      else
         System.out.println("Item not found or not enough units available.");
   }
   public void performAction(String command)
   {  if (command.equalsIgnoreCase("S"))
         System.out.println(this);
      else if (command.equalsIgnoreCase("A"))
         addUnits();
      else if (command.equalsIgnoreCase("D"))
         removeUnits();
   }
}
```

To make sure the class works properly, we can add a `main` method and execute it (see figure 3.61).

```
public static void main(String args[])
{  Menu menu = new Menu(TITLE + " Test", MENU, new Inventory()); }
```

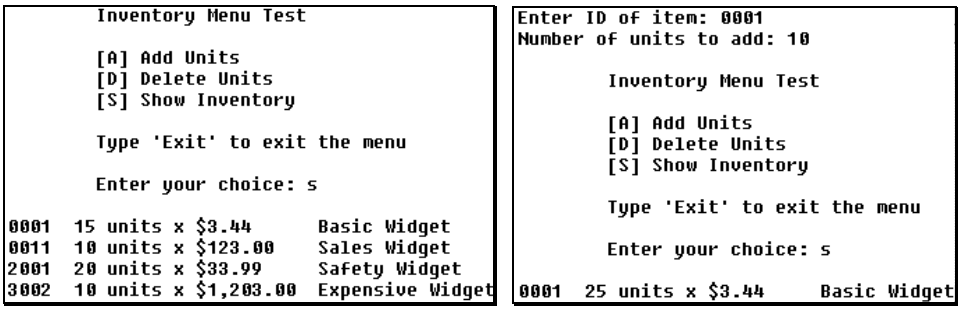


Figure 3.61: Testing `Inventory` *by selecting* `'s'` *(left) and* `'a'` *(right), adding 10 units to item 0001*

The next class to implement is `Invoice`. It obtains a `name` and a reference to an `Inventory` through its constructor. The `add` method uses `inventory.getPart` to ask the user for an inventory item and asks for the number of units to order. If the item is found and has that many units available, they are subtracted from the `InventoryItem` and a new `InvoiceItem` is added to the invoice list.

```
public class Invoice extends SortedList implements MenuUser
{  protected static final String MENU[] =
       {"[A] Add Item", "[D] Delete Item", "[S] Show Invoice"};
   protected static final String TITLE = "Invoice Menu";
   protected String name;
   private Inventory inventory;

   public Invoice(Inventory _inventory, String _name)
   {  super(new ItemComparer());
      inventory = _inventory;
      name = _name;
   }
   public void addItem()
   {  InventoryItem item = inventory.getPart();
      int units = Util.getInt("Units");
      if ((item != null) && (item.units >= units) && (units >= 0) && !isFull())
      {  item.remove(units);
         add(new InvoiceItem(item, units));
      }
      else
        System.out.println("List full, item not found or invalid units");
   }
   public void delItem()
   {  System.out.println(this);
      int index = indexOf(new Item(Util.getString("Enter item ID:")));
      if (index >= 0)
         delete(index);
   }
   public double getTotal()
   {  double total = 0.0;
      for (int i = 0; i < getSize(); i++)
         total += ((InvoiceItem)get(i)).total;
      return total;
   }
   public String toString()
   {  return "\n"+ name + ": " + Util.dollar(getTotal()) + super.toString(); }
   public void performAction(String command)
   {  if (command.equalsIgnoreCase("S"))
```

```
            System.out.println(this);
         else if (command.equalsIgnoreCase("A"))
            addItem();
         else if (command.equalsIgnoreCase("D"))
            delItem();
      }
   }
```

Finally we create the master class pulling everything together. All work is done by `Inventory`, `Invoice`, and `Menu`, so the master class is simple, as it should be in object-oriented programming.

```
public class AcmeInvoicer implements MenuUser
{  private final static String MENU[] = {"[I] Invoices", "[V] Inventory"};
   private Inventory inventory = new Inventory();
   private Menu menu = null;

   public AcmeInvoicer()
   {  menu = new Menu("Acme Invoicing System, Main Menu", MENU, this); }
   public void performAction(String command)
   {  if (command.equalsIgnoreCase("I"))
         menu = new Menu(Invoices.TITLE, Invoices.MENU,
                         new Invoice(inventory, Util.getString("Name")));
      else if (command.equalsIgnoreCase("V"))
         menu = new Menu(Inventory.TITLE,Inventory.MENU, inventory);
   }
   public static void main(String args[])
   {  AcmeInvoicer ai = new AcmeInvoicer(); }
}
```

Step 7: Test: We tested the `Inventory` class on its own (see figure 3.61) and it works properly. `Invoice` can be tested separately by adding the standard `main` method (see figure 3.62 d, e, f):

```
public static void main(String args[])
{  Menu menu = new Menu(TITLE, MENU, new Invoice(new Inventory(), "Test ")); }
```

Figure 3.62 shows some snapshots of executing the complete `AcmeInvoicer` program.

```
AWC Invoices, Main Menu

[I] Invoices
[V] Inventory

Type 'Exit' to exit the menu

Enter your choice: v
```

(a) `AcmeInvoicer` *main menu*

```
Inventory Menu

[A] Add Units
[D] Delete Units
[S] Show Inventory

Type 'Exit' to exit the menu

Enter your choice: exit
```

(b) `Inventory` *menu*

```
Invoice Menu

[A] Add Item
[D] Delete Item
[S] Show Invoice

Type 'Exit' to exit the menu

Enter your choice: a
```

(c) `Invoice` *menu*

```
Enter your choice: a

0001  15 units x $3.44       Basic Widget
0011  10 units x $123.00     Sales Widget
2001  20 units x $33.99      Safety Widget
3002  10 units x $1,203.00   Expensive Widget

Enter ID of item: 2001
Units: 2
```

(d) `Invoice` *menu, option 'a' to add 2 units of item 2001*

```
Enter your choice: a

0001  15 units x $3.44       Basic Widget
0011  10 units x $123.00     Sales Widget
2001  18 units x $33.99      Safety Widget
3002  10 units x $1,203.00   Expensive Widget

Enter ID of item: 0001
Units: 3
```

(e) `Invoice` *menu, option 'a' to add 3 units of item 0001*

```
Enter your choice: s

Test : $78.30
0001    3 units x $3.44  = $10.32  (Basic Widget)
2001    2 units x $33.99 = $67.98  (Safety Widget)
```

(f) `Invoice` *menu, option 's' to see the* `Invoice` *(sorted automatically)*

Figure 3.62: Testing the `AcmeInvoicer` *program*

Extensive testing reveals some shortcomings of our program:

- If an invalid integer is entered, `Console` halts program immediately. An improved `Console` class would fix that problem.
- If items with the same `ID` are entered in an invoice only the first one is found and deleted. Several solutions are possible: the user should adjust the number of units for an item instead of entering it twice, we could adjust `SortedList` class to return a 'sub-list' of items after searching, or we modify `SortedList` so that it would not accept duplicate entries.

We discuss these shortcomings with the owner and release the program for testing. The owner would like the additional features of modifying inventory descriptions and pricing, adding and removing inventory items, and storing invoices for later modification, otherwise everything works fine. We leave these features as an exercise.

Step 8: Document: We now embed documentation comments in our classes so that `javadoc` can create proper documentation. In particular, the `SortedList` and `Util` classes should be carefully documented because both can be useful to other projects. We also create a separate non-technical user manual for our program, listing in particular its shortcomings. The details are omitted, but class documentation and a user manual are essential pieces of a finished program.

Chapter Summary

In this chapter we introduced the following concepts and terminology:

Classes
See section 3.1, examples 3.01 (*Simple `Address` class*) and 3.02 (*Simple `Length` class*)

Objects and Instantiation
See section 3.1, example 3.03 (*Instantiating `Address` objects*)

Constructor
See section 3.1, examples 3.04 (*`Address` class with constructor*) and 3.05 *(Using `Address` as field types for `Parent` class*)

Garbage Collection, Destructor
See section 3.1, example 3.06 (*Creating a destructor method*)

Accessing Fields and Methods
See section 3.2, examples 3.07 (*Calling on display feature of `Address` and `Parent` classes*) and 3.08 (*Implementing and testing the `Length` class*)

Encapsulation using Private, Public, or Protected
See section 3.2, examples 3.09 *(Illustrating `private`, `public`, and friendly modifiers*), 3.10 (*Encapsulating the `Length` class*), 3.11 (*Creating a safe `Length` class*), and 3.12 *(A safe `Length` class with set/get methods*)

Utility Method:
See section 3.2, example 3.13 (*A safe `Length` class with utility methods*)

Class and Instance Fields
See section 3.2, example 3.14 (*Illustrating shared fields*)

Class Methods
See section 3.2, examples 3.15 (*Creating shared methods*) and 3.16 *(Using static fields to count instances of objects in memory*)

Overloading

See section 3.3, examples 3.17 (*Simple example using overloading*), 3.18 (*Presumed definition of* `System.out.println`), 3.19 (*Overloading static methods*), 3.20 (*Overloading constructor for* `Student` *class*), and 3.21 (*Using overloading to compute total course GPA*)

Inheritance

See section 3.4, examples 3.22 (*Extending the* `Address` *class*) and 3.23 (*Creating* `Shape` *classes using inheritance*)

Implicit Call of Superclass Constructor

See section 3.4, examples 3.24 (*Illustrating implicit call of superclass constructor*) and 3.25 (*Adding a* `Square` *to the* `Shape` *classes*)

Overriding Methods and Fields

See section 3.4, examples 3.26 (*Overriding the* `display` *method in the* `Shape` *hierarchy*) and 3.27 (*Illustrating usefulness of using* `super` *while overriding*)

Abstract Methods and Classes

See section 3.4, examples 3.28 (*Abstract methods for* `Shape` *classes*) and 3.29 (*A menu-driven* `CheckBook` *program using abstract methods*)

Final Methods and Classes

See section 3.4, example 3.30 (*Simple billing system using final fields and abstract methods*)

The Object Class

See section 3.5, examples 3.31 (`Object` *as superclass for* `Shape` *hierarchy*), 3.32 *(Overriding the* `toString` *method*), 3.33 (*Defining an array of* `Object` *to store different types*), and 3.34 (*A general* `List` *class*)

Multi-Level and Multiple Inheritance

See section 3.6, examples 3.35 (*Inheritance type of* `Shape` *hierarchy*) and 3.36 (*Implicit call of superclass constructor in multi-level hierarchies*)

Interfaces

See section 3.6, examples 3.37 (*Interfaces as a new type*), 3.38 *(Sorting arrays using a* `Sorter` *class and* `Sortable` *interface*), 3.39 (*Creating a general-purpose menu system using classes and interfaces*), and 3.40 (*Sorting arrays using a* `FlexSorter` *class and* `Comparable` *interface*)

Polymorphism

See section 3.6, examples 3.41 (*Illustrating polymorphism*), 3.42 (*Using polymorphism to call* `toString`), 3.43 (*Using polymorphism for* `Shape` *hierarchy*), and 3.44 (*Adding* `RightTriangle` *to* `Shape` *hierarchy*)

Case Study: OO Design for Invoice Program

1.