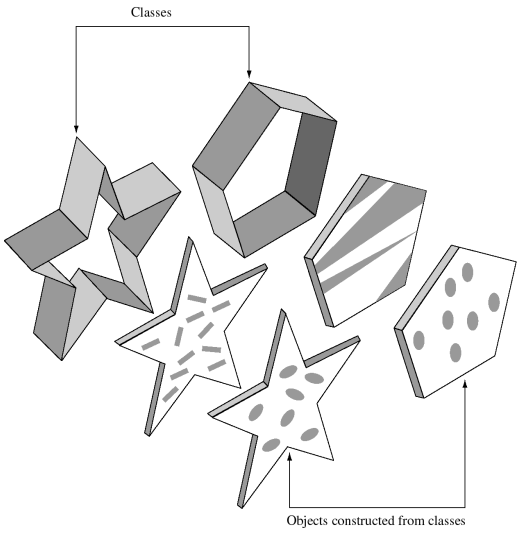
Object-Oriented Programming with Java
Edwin J. Kay and Glenn D. Blank

Chapter 2

Object orientation

Each computer programming language has an underlying metaphor for its organization and for the way in which a program written in the language articulates the solution to a given problem. In the metaphor for object-oriented programming languages tasks are accomplished by *objects* that communicate with other objects by sending them *messages*. An object contains information (*data*) about its current state and responds to a message by sending other objects messages and then possibly returning (responding with) either a value, e.g., a number, or an object. Thinking of me as an object and a calculator as a second object, if someone sends me the message "Please add up this column of numbers" I might "send a series of messages" to the calculator (key some instructions into the calculator) instructing it to add up the numbers. The calculator responds to each "message" I send it by displaying the result on its screen. We can think of that display as the value returned by the calculator after each message. I, in turn, respond to the person who sent me the message by telling her the sum of the column of numbers.

In Java, the objects are organized into *classes*. Each object in a class is called an *instance* of that class. So, for example, if I have a class Calculator, it should be possible to create many instances, such as myCalculator and yourCalculator. All instances of a class contain the same kind of information and can perform the same behaviors. The relationship between a class and an object or instance is rather like that of a cookie cutter and cookies. Like a cookie cutter, a class is a shape, with which creates many individual objects. A **class** is a shape, an *abstraction*, which defines the data and behaviors of that class. One can then create a batch of particular objects or instances, all of which are separate entities, but share the same properties and functionality.



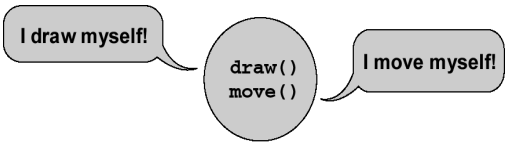
Classes as cookie cutters

All objects (instances) of a class include the same data, called *instance variables*. The class describes the instance variables, and whenever we create an instance of the class, it will contain its own space for the instance variables. For example, the class Calculator describes an instance variable called Accumulator. Each instance will contains its own instance of Accumulator. Each instance variable can store a different value. If I tell myCalculator to add 2 to itself, that will just change the value of myCalculator's Accumulator. It will not affect yourCalculator's Accumulator, nor that of any other instance.

For each message to which an object responds, there is a set of steps that the object follows to make the response. We can think of the object following a set of instructions that specify the steps followed. In Java, this set of instructions is called a *method.* A class defines a set of methods. If I

send a message to myCalculator, it will execute the corresponding method, defined in class Calculator. In Java, instances of a class send messages to an object using a "dot" notation. For example, myCalculator.add(2) asks myCalculator to "add 2 to yourself", which myCalculator by executing the corresponding method.

Note the way we are talking to objects. If want a circle object, I send a message to it: draw yourself on a screen, or in Java, aCircle.draw(). Part of the culture of object-orientation is that objects take on a little life of their own; they are more *autonomous*. To say that an object "draws itself" implies that its behaviors are part of its nature. In more traditional, procedural terms we might issue a command: "draw a circle"—which implies that behaviors are independent of what they act upon. Some object-oriented practitioners even advocate "object think" as a habit of mind: think of objects as if they were *agents*, relatively independent of other objects, though perfectly willing to cooperate. For example, pretend a Circle object can announce to the wide world, "I draw myself!" and "I move myself!" Thinking and talking about objects as if they were autonomous agents can help you visualize how they interact with each other. Try it!



If objects could talk!

In Java, the class itself is an object. Its data include the instances of the class. Initially, the class starts out with no instances, but the class can respond to messages to create instances. As on object, the class's most important member method is its *constructor*, which is used to create (*instantiate*) instances of the class. Later we will see that a class can have other kinds of data besides its instances and other kinds of methods besides a constructor.

Exercise 2.1: Why does Java call the data of an object instance variables? Explain your answer in terms of how objects are created and where its data are stored.
Exercise 2.2: What is the relationships between messages and methods?
Exercise 2.3: How do you feel about attributing agent-hood to software objects? Do you think it's a good idea to anthropomorphize computer programs or objects (a program is "thinking" or Knobby is "talking" or an object is "moving myself")? Or do you agree with some computer scientists who argue that this tendency just encourages sloppy thinking?

A bit of "object think"
Let's consider a few examples classes of objects. First, consider a hand calculator. Of course there are many makes and models of hand calculators, each make and model having somewhat different properties. To simplify the discussion and to facilitate the abstraction of the calculator, we assume there is only one manufacturer of hand calculators, and we assume that the manufacturer only makes one model. (Java does have structures that allow us to have, e.g., more than one kind of calculator.) Given these simplifying assumptions we can think of the class Calculator as consisting of all hand calculators, where each calculator (instance of a Calculator) has the same data and the same operations (methods) on the data. We can think of sending the calculator a "message" to store a number. It responds this message by displaying ("returning") on the screen the number stored. We can think of sending a message to the calculator to add a number to the number stored. We can think of sending a message to the calculator to take the square root of the number stored, etc.

For our second example, consider a shopping list. Our first example, by simplifying the real world notion of a calculator, hints at how we abstract real world things as "objects." When we consider the idea of a shopping list we again must abstract the idea to think of it as an object. We make decisions about what are important aspects of a shopping list as we design an object to abstract the idea of a shopping list. In our design, objects in the class ShoppingList have, as data members, a collection of items and their prices. ShoppingList objects have methods for adding items to the collection, for setting the prices of items that have been stored, for getting the price of an item, for getting the total cost of the items whose prices have been set, and for determining whether a given item is in the list. Of course the list of methods is incomplete; we have failed to abstract a number of other properties of a shopping list.

Next, let's consider three interrelated examples of classes, based on the cyclometer that is on my bicycle. This cyclometer keeps track of the amount of time my front wheel rotates and how many revolutions occur when it does rotate. It then provides me with a great deal of information at the push of a button (I suppose on the theory that bicycling is a mindless activity that needs to be enhanced with a glitzy display). It determines how fast I am riding, how far I have gone, the maximum speed I have gone, etc. In trying to design a class to abstract this device, I start with the class Cylcometer. The cyclometer on my bicycle gets sent an electronic message each time a little magnet attached to a spoke of my front wheel goes past a sensor attached to the front fork of my bicycle. The cyclometer has an internal clock and some other devices that help it make sense out of this information. But more of that later. For the moment, we only need to know that we send the cyclometer electronic messages every time the magnet goes past the sensor. We have to send it a message to store the diameter of the front wheel so that it can carry out various calculations for displaying information on the screen. We also send it messages requesting the speed, the distance traveled, the average speed, and the maximum speed. In practice I do this by pushing a button which changes the display on my cyclometer's screen. From the point of view of the messages a cyclometer responds to, the class is quite simple, but the data members are sufficiently complicated that we represent them as instances of classes as well: a Clock class to represent the clock and a Counter class to represent the counters, one that keeps track of how much time elapses when the wheel is rotating and one that keeps track of the number of wheel rotations. In this view the main job of an instance of the class Cyclometer consists of moving information among its various data members. One of its data members stores information that indicates whether the wheel is rotating. If the wheel is rotating, it uses the clock to increment the time counter every second (or millisecond if more accuracy is needed). At the same time it increments the rotation counter after each rotation of the wheel.

The class Counter is straightforward. An instance of Counter should enable messages for resetting it to zero, for incrementing it, and for obtaining the number of counts that have been accumulated.

The class Clock is straightforward as well. When an instance of clock is created it starts ticking, keeping track of the number of elapsed seconds (or milliseconds). It responds to a message requesting the time by returning the number of elapsed seconds. (In a fancier version of class Clock instances would be synchronized with real time and respond to messages requesting the time of day in hours, minutes, seconds, etc., but we don't need such a fancy version.)

To see how an instance of Cyclometer could use the information from its data members to compute the velocity, for example, see the text box to the right.

> Assume that the wheel diameter is 27 inches, that the time counter has counted 400 seconds, and that the revolution counter has counted 940 revolutions. Then the distance covered is 940 times the circumference of the wheel (27" x pi = 84.82"): 84.82" x 940 = 79733.55". This is 79733.55/(5280*12) = 1.2584 miles. The average speed is 1.2584 miles per 400 seconds or (1.2584/400)*3600 = 11.33 miles per hour.

Although I have described the three classes Cyclometer, Counter, and Clock in some detail, I have not implemented them. I may have missed some subtle problems with the design of the class Cyclometer, especially in the way in which it uses the classes Counter and Clock. Implement the classes in Java and then testing will likely reveal any problems that I had not thought about. I am sure I could repair any subtle problems that I uncover.

How did the makers of my cyclometer design and build it anyway? I don't know for sure but assume that like most manufacturers, they wrote a computer program to simulate the product before building it. This enables them to better understand how it works and how user friendly it is. It is quite easy to change the program when design changes are called for. If they actually build the physical device and then discover design flaws they would have to through away the device and start again. Building the physical device too early could lead to quite a trail of rubbish, whereas building it in software leaves no rubbish at all.

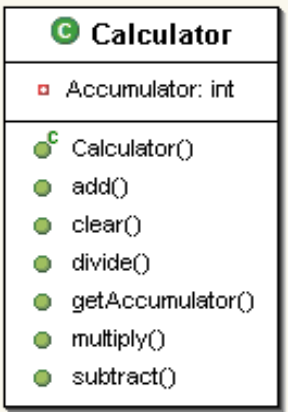
In talking about designing class Cyclometer I have implicitly been using some design principles. First, I started with the design of the Cyclometer and then went on to design the Counter and Clock classes after I understood how their instances would be used. This is a *top-down* approach. A bottom-up approach, in which I design the classes used by Cyclometer before designing the class Cyclometer is less likely to work well, because I may not provide the classes Counter and Clock with the correct functionality. Second, instead of having instances of class Cyclometer do all the work, I parceled out the work to instances of Counter and Clock. Thus, I decomposed the larger problem into smaller, more manageable components.

We can think of a class itself as object whose main job involves producing instances of the class. The class's data include the instances of the class. Initially, the class starts out with no instances, but the class responds to messages to create instances. As on object, the class's most important member method is its *constructor*, which is used to create (*instantiate*) instances of the class. Later we will see that a class can have other kinds of data besides its instances and other kinds of methods besides a constructor. For now, it is important to remember that in describing a class we describe the behavior of the class as an object in terms of how it constructs its instances, and we describe the behavior of the instances of the class.

The class Calculator example in Java

We will now go over the first two examples again, this time making limited use of Java syntax to describe the way the classes and their instances behave. In general, *syntax* is the set of rules that govern the order of words in a language. The sequence of words "boy ball hit a the" (probably) violates (the rather loose) syntax rules for English and makes no sense, although you can readily

rearrange the words into a syntactically correct English sentence. By Java syntax we mean the rules governing the ways in which one may legitimately describe an object, the object's data members, and the object's member methods in Java. Java syntax rules are very strict; a minor violation of the rules will produce an unacceptable Java program. The full set of syntax rules for Java is long and complicated, but one can readily write Java programs with a smaller subset. Subsequent chapters will present some of these rules. For now, we'll start covering just enough Java syntax to create an instance of a class and then send it messages using its methods.



At left is a UML diagram of class Calculator. Instances store numbers in the Accumulator and have various methods for doing arithmetic and retrieving the contents of the Accumulator. A UML diagram depicts a class in three parts: 1) a *name* (in this case Calculator) in the top section, a set of *attributes* (in this case, there's just one, Accumulator) in the middle section, and a set of *operations* in the bottom section. Each attribute has a *type*, describing the kind of data it can store; in this case Accumulator belongs to type int, or integer. You'll notice that UML and Java have different ways of describing the same things. That's because other object-oriented languages use other jargon, and UML is independent of any particular programming language. What UML calls an attribute, Java calls an instance variable. What UML calls an operation, Java calls a method. Got it?

Later we will explain the exact details of how to write Java code for creating the class Calculator. Once we have the Java class Calculator, we can then give the class instructions by sending it messages. Each of these messages is specified with a Java *statement*. We now show the Java statements that could be used to create an instance of a Calculator and then send it messages using its methods. First we need to have a statement that gives a name to the instance, say

```
Calculator aCalc;
```

Here we have *declared* a *variable* aCalc to be of type Calculator. Thus it will be used to represent (*reference*) an instance of Calculator. Next we create an instance to which aCalc will refer:

```
aCalc = new Calculator();
```

Here we have used the word "new" to call upon the constructor for class Calculator to create a new instance of a Calculator. Note that if we repeat the statement

```
aCalc = new Calculator();
```

we are creating another new instance of a Calculator and discarding the previous instance. Also note that the two statements

```
Calculator aCalc;
aCalc = new Calculator();
```

can be combined into a single statement

```
Calculator aCalc=new Calculator();
```

Further note that semicolons (;) are used to indicate the ends of statements.

Now that we have an instance of Calculator we can send it messages by calling (*invoking)* its methods (the names of which we somehow know). First we clear the calculator.

```
aCalc.clear();
```

Pay attention to the syntax of the above statement. The dot (.) notation indicates we are accessing a data member or member method of aCalc. In this case, we are accessing the method clear(). We

know it is a method because of the parentheses. The parentheses of a method enclose zero or more *arguments* (or *parameters)*, separated by commas, that determine the behavior of aCalc in response to the message. Because clear() takes no arguments, the behavior of aCalc in response to clear() is always the same. Now, we add the number 42 to the calculator and then multiply by 3.

```
aCalc.add(42);
aCalc.multiply(3);
```

The methods add() and multiply() each take one argument (parameter), the number to be added or multiplied. We now send the calculator the message to return the value that is currently stored in the calculator.

```
aCalc.getAccumulator();
```

The calculator returns the value, but in this statement we are ignoring it. To display the contents of the calculator we can write

```
System.out.println("aCalc value: " + aCalc.getAccumulator());
```

This last statement is a fairly complicated one. Though it's not really necessary to understand it in detail right now, the box to the right provides a detailed explanation.

> System is an object that handles input and output. The object out in System handles output, which we access via the dot notation. The object System.out has a method println(), which takes one parameter. The parameter is a Java *string* (a sequence of characters which we denote by surrounding them with quotes and which can be added to other strings by using the + operator). We provide a string argument to println() from two parts, put together with The + operator: the string "aCalc value: " and the integer that aCalc.getAccumulator() returns. As a result of all of this, the following appears on the screen:
>
> aCalc value: 126

Continuing the example, we write a sequence of statements that create a calculator, clear it, add 2 to it, display the result on the screen, multiply by 12, display the result on the screen, divide by 5, and display the result on the screen.

```
Calculator y=new Calculator();
y.clear();
y.add(2);
System.out.println("y value: " + y.getAccumulator());
y.multiply(12);
System.out.println("y value: " + y.getAccumulator());
y.divide(5);
 System.out.println("y value: " + y.getAccumulator());
```

This code displays the following on the screen:

```
y value: 2
y value: 24
y value: 4
```

To finish the example, we write some code that at first glance does not make much sense, that at second glance uses the above ideas in an unexpected way, and that at third glance might plausibly work. Indeed, all statements, including the second to last statement below, do work.

```
Calculator a=new Calculator(), b=new Calculator(), c=new Calculator();
a.clear();
b.clear();
```

```
c.clear();
a.add(34);
b.add(45);
c.add(a.getAccumluator() + b.getAccumulator());
System.out.println("c value: " + c.getAccumulator());
```

This code displays the following on the screen:

```
c value: 79
```

The first statement above provides a compact way for creating instances of Calculator, the three instantiations separated by commas.

All of the above statements make the assumption that the messages are sent (the methods execute) in sequential order, i.e., the order in which they appear. You quite likely made the same assumption when reading the above examples but it is useful to make this assumption explicit.

An *identifier* is the name of a class, method or variable. In Java, an identifier is a sequence of upper case letter (such as A or G) , lower case letters (such as b or x,)underscores (_), dollar signs ($), or digits, except that it may not start with a digit. (Why not start with a digit? To avoid ambiguity, since digits can start numbers.) It is important to remember that Java distinguishes between upper and lowercase letters for these names. Thus, getAccumulator() and getaccumulator() are the names of different methods. By convention, we start the names of classes with an uppercase letter; we start the names of member methods and instance variables with lowercase letters. Often the name of a variable is a short phrase (without blanks, e.g., aCalc). In this case we start each word in the phrase after the first with an uppercase letter, e.g., totalTestScore. You cannot use *keywords*, such as int and new, as identifiers, because they are *reserved* as part of the Java language.

Instances of the class Calculator clearly have to store the results of its calculations someplace. However, this value is not directly accessible, whereas the attribute "out" in class System is accessible. In the same way, the methods of an object may or may not be accessible. There is a good reason for making values and methods inaccessible. It stops the person using the objects from meddling with them and causing some problem.

We can think of an object as being in a particular state, depending on the specific values stored in its instance variable(s). Typically, messages lead to changes in the state of the object. For example, the state of a Calculator is the value stored in its Accumulator.

Exercise 2.4: Write a snippet of Java code that creates another instance of Calculator called myCalc. Then ask myCalc to clear its Accumulator, subtract 4, multiply by 2, and display the result. What output would you expect your snippet to display?

Exercise 2.5: Write a snippet of Java code that creates two instances of Calculator, clears each calculator, has the first calculate the product of 10 and 20, has the second calculate 200 divided by 6, and then displays the sum of the contents of the two calculators on the screen.

Exercise 2.6: What's wrong with following snippet? Fix it so that it produces the desired behavior.

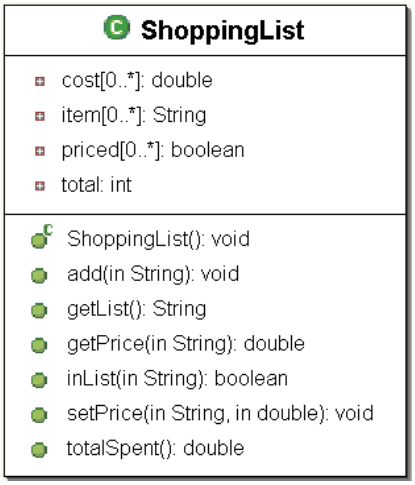
```
System.out.println(Calculator.getAccumulator());
```

Exercise 2.7: Which of the following are valid Java identifiers? Explain why or why not.

a) IRS b) Internal Revenue c) IRS$ d) R2D2 e) 3CPO
f) int g) I.R.S. h) X_men i) X-men j) $moolah k) shalom!

The class ShoppingList example

Now we give a second example (with a bit less explanation) of a Java class representing an object. In this case we are thinking of a shopping list as the object. We want to be able to enter items in the list, to set the price of items in the list to indicate we have bought (or are about to buy) them, to get the cost of an item on the list, to determine whether an item is in the list, to determine the total spent on items that have been priced, and to get a display of the list. We depict class ShoppingList in the figure on the left. The first three instance variables—cost, item and priced—each stores a list (or array) of zero or more items. The cost list stores real numbers, each of which Java represents as a double. The item list stores the names of the item, each of which Java represents as a String. And the priced list stores true/false values (each of which Java represents as a boolean) indicating whether the corresponding item in the item list has a price or not. The operators include a constructor, ShoppingList(), and five methods that access or modify the values of the instance variables. You can also see the type of data each operator returns (void means nothing) as well as what it expects to receive (e.g., "in String").[1]



Below is some code using the ShoppingList class. Beside various statements are explanatory comments. In Java programs comments are preceded by two forward slashes (//). Anything following two forward slashes on a line is ignored in Java.

```
ShoppingList sl=new ShoppingList();   //call shopping list's constructor; start with an empty list
sl.add("Jello");
sl.add("Granola");
sl.add("Froot Loops");
sl.setPrice("Jello",2.25);
sl.setPrice("Junk",19.95);            //by design, nothing happens, because "Junk" is not in the list
sl.setPrice("Granola",1.79);
System.out.println(sl.getList()+      //display the list
      "The total cost so far is " +   //followed by a message describing what follows
      sl.totalSpent()  );             //followed by the total amount spent
System.out.println("It is " + sl.inList("Granola") +   //inList() returns either "true" or "false"
   " that Granola is in the list\n" +                  //in Java, \n means "go to the next line"
   "Granola costs $" + sl.getPrice("Granola"));
```

Running the above code produces the following output:

[1]Note that the UML diagram shown for ShoppingList (unlike the one for Calculator above), shows parameters and parameters. To get this look (which we recommend) in Eclipse, right-click on a class name in a box, then select "View selector" from the menu, then change the way methods appear.

```
Shopping List
------------------
Jello   $2.25
Granola   $1.79
Froot Loops
----------------------------
The total cost so far is $4.04
It is true that Granola is in the list
Granola costs $1.79
```

Try to follow the code to see how it produces the output. Note that the method getList() returns a string that starts with "Shopping List", continues with "----------", then lists the items and their costs, and finishes with "----------". In designing the class, there are a number of issues that need to be dealt with in practice. What happens when you try to add an item that is already there? Is the item "Granola" and the item "Granola " the same item (the latter string having a space at the end)? What happens when you try to retrieve the cost of a non-existent item? Also, one might want to have other methods. For example, as conceived so far, the class ShoppingList does not have a method for removing an item from the list, but it probably should.

Exercise 2.8: From the above code (and comments), can you explain the difference between the behavior of sl.setPrice("Jello",2.25) and sl.setPrice("Junk",19.95)? Why is this difference a reasonable design decision?

Exercise 2.9: The UML diagram for ShoppingList shows the types of data that methods receive between parentheses and the type of data it returns after the colon. What type of data that setPrice() expect (between parentheses)? Give an example from the above code. What type of data does getPrice() take and what does it produce? Why is it important to consider this kind of information?

Exercise 2.10: The '+' operator seen several times in the code above is called a *concatenation* operator. Explain in your own words what you think this operator does. Why do think Java would do if it saw the expression, 2.25 + 19.95? How would Java know what to do?

Exercise 2.12: Write Java code that creates an instance of a ShoppingList, adds "hamburger", "french fries" and "shake" to the list, sets the price of each item to a reasonable amount, determines whether we ordered a "cheeseburger" then a "hamburger", and finally displays the total cost.

Exercise 2.13: Write Java code that creates an instance of a Calculator and an instance of a ShoppingList, adds "Shoes", "Socks", "Pants", and "Shirt" to the list, sets the price of "Shoes" to $70, displays the cost of "Shoes", displays the list, sets the price of "Shirt" to $32.50, displays the list, retrieves the cost of the "Shoes" and "Shirt" and uses the calculator to add them, displays the results from the calculator, and then displays the total cost of the shopping list (which should agree with the results from the calculator).