Preface to 1st Edition

Control abstraction was the message of the first programming revolution seen in high-level programming languages such as Algol and Pascal. The focus of the next revolution was data abstraction, which proposed languages such as Modula and Ada.

The object-oriented revolution began slowly in the 1960s with the programming language Simula, but moved onto more languages such as Smalltalk, Objective-C and C++. Java is almost a hybrid between Smalltalk and C++, and has gained widespread acceptance due to its association with the Internet, its availability to a large user base and reusable libraries for programming in a graphical environment.

Our programming lineage has passed through Pascal, C and C++. As with many other programmers, good run-time checks with automatic memory management and a reusable API made Java a very attractive option. After a half-day on the original Java Whitepaper and the early Java online tutorial, we were sold on the Java bandwagon and already writing code. In another two days' time, we were using the Abstract Windowing Toolkit (AWT) package for graphical applications. In situations where there is no large investment into older languages, we are quite happy to abandon them completely.

Effective programming in Java comes from understanding three key areas – object-oriented concepts, the syntax and semantics of the Java programming language and the Java Application Programming Interface (API). This is our emphasis when we conduct professional courses, and in this book as well.

Much of the material in this book is based on previous courses which we have conducted over the past two years to the industry and the National University of Singapore (NUS). Courses conducted for the industry last about 5 to 7 days, depending on the amount of coaching that participants require. In the Department of Information Systems and Computer Science at NUS, a course on "Object-Oriented Methods" runs over 13 weeks.

As you might have noticed, we have taken to Java as ducks to water. Java has allowed us to think about and specify object behavior. This results in executable code which is merely secondary. What is important is the clean specification of object behavior. Similarly, in getting accustomed to working with objects, we believe that you will enjoy it too.

Preface to 2nd Edition

Since publishing the first edition almost 10 years ago, we have seen Java being used in many high school and university programming courses. Further, many projects now use Java as the implementation language. Similarly, at the Institute of Systems Science, we have seen professional developers warming up to Java for the first time in 1998, to those who use Java in their daily work in 2007.

We have thus updated the material to cover J2EE topics such as JDBC, RMI, Serialization and Java Servlets. We have also added a chapter on Generics as the Java language evolved to allow this elegant feature.

For those who might be embarking on a Java journey now, we wish you a pleasant journey and a well-used road map. Many have taken this journey before and are enjoying the fruits of their learning investment.

2

Object, Class, Message and Method

We had our first introduction to objects, message and method in Chapter 1. Another concept closely associated with the concept of objects is *class*. In object-oriented programming, a *class* is a definition template for structuring and creating objects.

In this chapter, we will discuss the concept of object, message, method and class and how these concepts are used in a computer model.

2.1 Objects and Class

In Chapter 1, we introduced Benjamin. Now, meet Bernie, another customer at HomeCare. As customers of HomeCare, Benjamin and Bernie share some similar information. For example, both have a name, an address, and a budget—information that is relevant when describing customers. This information is known as *object attributes*.

An object attribute definition allows for objects to have independent *attribute values*. For example, Benjamin may have a larger budget and thus a larger budget value (say $2000) than Bernie whose budget may be $1000. Collectively, the values of an object's attributes represent the *state* of the object.

Besides attributes, Benjamin and Bernie also exhibit some behavior typical of a customer. For instance, Benjamin and Bernie execute a *method* when making a purchase. Let us call this method `purchase()`. The method `purchase()` is made up of a set of operations that Benjamin and Bernie would use to send a purchase request to a salesperson.

Structurally, Benjamin and Bernie can be represented as follows:

```
Benjamin as an Object
  Attributes:
    name = "Benjamin"
    address = "1, Robinson Road"
    budget = "2000"
  Methods:
    purchase()   {send a purchase request to a salesperson}
    getBudget()  {return budget}
Bernie as an Object
  Attributes:
    name = "Bernie"
    address = "18, Sophia Road"
    budget = "1000"
  Methods:
    purchase()   {send a purchase request to a salesperson}
    getBudget()  {return budget}
```

`name`, `address` and `budget` are attributes while `purchase()` and `getBudget()` are methods of the two objects. Note that both objects share a common definition of attributes and methods. In fact, all customers of HomeCare share the same set of attribute and method definitions. They all have attributes `name`, `address` and `budget`, and methods `purchase()` and `getBudget()`. In defining these objects, a common definition known as *class* is used.

A class is a definition template for structuring and creating objects with the same attributes and methods. Benjamin and Bernie, being customers of HomeCare, can therefore be defined by a class called Customer as follows:

```
Class Customer
  Attributes:
    name
    address
    budget
  Methods:
    purchase()   {send a purchase request to a salesperson}
    getBudget()  {return budget}
```

One major difference between objects and class is in the way attributes and methods are treated in objects and classes. A class is a definition about objects; the attributes and methods in a class are thus declarations that do not contain values. However, objects are created instances of a class. Each has its own attributes and methods. The values of the set of attributes describe the state of the objects.

Let us now examine the salespersons. Salespersons also have attributes and methods. Sean and Sara are two salespersons at HomeCare. They are thus capable of a behavior typical of a salesperson, for example, taking orders from customers. To fulfill their role as salespersons in a purchase transaction, Sean and Sara perform a method. We shall call this method `takeOrder()`, and represent Sean and Sara as follows:

```
Sean as an Object
  Attributes:
    name = "Sean"
  Methods:
    takeOrder()      {
      check with warehouse on stock availability
      check with warehouse on delivery schedule
      if ok
      then {instruct warehouse to deliver stock(address, date)
            return ok}
      else return not ok
    }

Sara as an Object
  Attributes:
    name = "Sara"
  Methods:
    takeOrder()      {
      check with warehouse on stock availability
      check with warehouse on delivery schedule
      if ok
      then {instruct warehouse to deliver stock(address, date)
            return ok}
      else return not ok
    }
```

Being salespersons, Sean and Sara share similar attributes and methods as expected. Like the customers, their definition can be described by a class called SalesPerson with the following representation:

```
Class SalesPerson
  Attributes:
    name
  Methods:
    takeOrder()      {
      check with warehouse on stock availability
      check with warehouse on delivery schedule
      if ok
      then {instruct warehouse to deliver stock(address, date)
            return ok}
      else return not ok
    }
```

Note that the definition of the SalesPerson class is different from the Customer class since customers and salespersons behave differently—customers make orders and salespersons take orders.

2.2 Message and Method

Objects communicate with one another by sending *messages*. A message is a *method call* from a message-sending object to a message-receiving object. A message-sending object is a *sender* while a message-receiving object is a *receiver*.

An object responds to a message by executing one of its methods. Additional information, known as *arguments*, may accompany a method call. Such parameterization allows for added flexibility in message passing. The set of methods collectively defines the dynamic behavior of an object. An object may have as many methods as required.

2.2.1 Message Components

A message is composed of three components:

- an object identifier that indicates the message receiver,
- a method name (corresponding to a method of the receiver), and
- arguments (additional information required for the execution of the method).

Earlier we saw that Benjamin sent a message to Sean when Benjamin wanted to buy a sofa set. The reasonable location for Benjamin to send the message to Sean is in Benjamin's `purchase()` method as shown below (indicated in bold):

```
Benjamin as an Object
  Attributes:
    name = "Benjamin"
    address = "1, Robinson Road"
    budget = "2000"
  Methods:
    purchase() {
      Sean.takeOrder("Benjamin", "sofa", "1, Robinson Road",
                     "12 November")
    }
    getBudget()      {return budget}
```

The message `Sean.takeOrder(who, stock, address, date)` is interpreted as follows:

- Sean is the receiver of the message;
- `takeOrder` is a method call on Sean;
- `"Benjamin","stock","address","date"` are arguments of the message.

2.2.2 Method

A message is valid if the receiver has a method that corresponds to the method named in the message and the appropriate arguments, if any, are supplied with the message. Only valid messages are executed by the receiver. The `takeOrder()` message is valid because Sean has a corresponding method and the required arguments (`who`, `stock`, `address`, `date`) are supplied with the message.

Sean's `takeOrder()` method is made up of a set of operations (indicated in bold below) as follows:

```
Sean as an Object
  Attributes:
   name = "Sean"
  Methods:
    takeOrder(who,stock,address,date)   {
      check with warehouse on stock availability
      check with warehouse on delivery schedule
      if ok then {
        instruct warehouse to deliver stock to address on date
        return ok
      } else return not ok
    }
```

In the above description, a message is sent from Sean to a Warehouse object to inquire on the order and delivery schedule in Sean's `takeOrder()` method. If both conditions are satisfied, Sean will instruct the Warehouse object to arrange for delivery.

How Sean carries out the method is known only to Sean. Neither Benjamin nor the other customers know how Sean does it. For example, to check on the stock and delivery schedule with the warehouse, Sean may have called the warehouse over the phone or he may have checked the information against a list he had gotten from the warehouse. What Benjamin knows of Sean is that Sean is capable of responding to his request since his message to Sean is acceptable by Sean.

In object-oriented programming, Benjamin and Sean are said to have followed the principle of *information hiding—How* Sean is going to satisfy Benjamin's request *is hidden* from Benjamin. In this way, Sean is free to select whatever way he chooses to satisfy Benjamin's request; he may phone the warehouse or look up the pre-prepared list and vice versa.

2.2.3 Client and Server

By executing a method, a message-receiving object (such as Sean) is said to serve the message-sending object (such as Benjamin). A message-receiving object is thus a *server* to a message-sending object and the message-sending object is thus a *client* of the server.

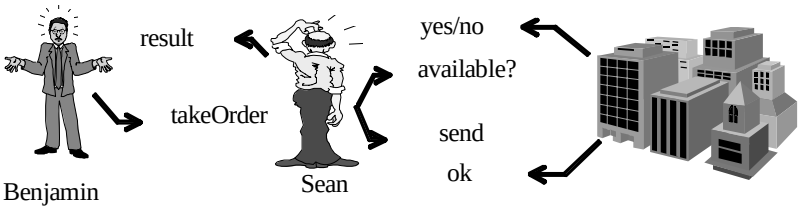

Figure 2-1: Object communication process.

In any object communication, there are at least a client and a server. The client sends a message to request a server to perform a task. The task is fulfilled by a

message-corresponding method of the server. In sending a message to the warehouse, Sean is said to be the client and the warehouse is said to be the server.

Benjamin, Sean, and the warehouse are three objects involved in a communication process. Benjamin is the initiator, with Sean and the warehouse as partners in the communication process. Figure 2-1 depicts a typical communication process amongst objects.

2.3 Creating Objects

In object-oriented programming, objects are created from classes. Instances of Customer objects are created from a Customer class and SalesPerson objects from a SalesPerson class.

Created object instances are individuals with their own state. To illustrate, let us consider the example of counters. A counter is a device that keeps account of the number of times an event has occurred. It has two buttons: an initialize button that resets the counter to 0, and an add button that adds 1 to its present number. Figure 2-2 shows a counter with a number 10.

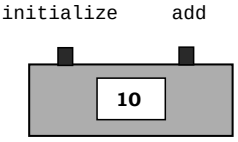

Figure 2-2: A counter.

Structurally, the first counter object can be represented as follows:

```
First Counter Object
  Attributes:
    number = 10
  Methods:
    add()            {number = number + 1}
    initialize()     {number = 0}
    getNumber()      {return number}
```

Figure 2-3 shows two more counters.

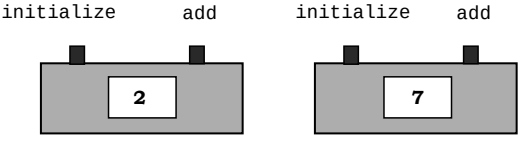

Figure 2-3: Two Additional Counters

Like the first counter, these two counters may be reset to zero and incremented through the `initialize` and `add` buttons respectively, and represented as follows:

```
Second Counter Object
  Attributes:
    number = 2
  Methods:
    add()            {number = number + 1}
    initialize()     {number = 0}
    getNumber()      {return number}

Third Counter Object
  Attributes:
    number = 7
  Methods:
    add()            {number = number + 1}
    initialize()     {number = 0}
    getNumber()      {return number}
```

All the three counters share the same definition of attributes and methods, and like in the previous examples, they can be defined by a class as follows:

```
Class Counter
  Attributes:
    number
  Methods:
    add()        {number = number + 1}
    initialize() {number = 0}
    getNumber()  {return number}
```

The Counter class has:

- an attribute, `number`;
- an `initialize()` method that causes a counter to reset its `number` to 0.
- an `add()` method that causes a counter to add 1 to its `number`; and
- a `getNumber()` method that returns the current value of the attribute `number`.

Suppose a new object is created from the Counter class. Although the new Counter object would have the same definition of attributes and methods as the previous three counters, its attribute value may not be the same as the other counters. This suggests that the state of the counters may be different from one another.

For the newly created fourth Counter object, it has a state represented by the attribute `number` with a value of `0`, the value at initialization:

```
Fourth Counter Object
  Attributes:
    number = 0
  Methods:
    add()        {number = number + 1}
    initialize() {number = 0}
    getNumber()  {return number}
```

Note that the attribute value of the fourth Counter object is different from the other three counters.

2.4 Summary

In this chapter, we discussed:

- Objects are defined by classes.
- Objects from the same class share the same definition of attributes and methods.
- Objects from the same class may not have the same attribute values.
- Objects from different classes do not share the same definition of attributes or methods.
- Objects created from the same class share the same definition of attributes and methods but their state may differ.
- A method is a set of operations executed by an object upon the receipt of a message.
- A message has three components: an object identifier, a method name and arguments.
- A message-receiving object is a *server* to a message-sending object known as a *client*.

2.5 Exercises

1. Distinguish the terms “Object” and “Class”.
2. Consider the scenario of buying flowers from a florist. Outline the objects in such a transaction together with the messages exchanged.
3. Given a class definition Rectangle below, describe the structure of any 3 instances of Rectangle.

```
class Rectangle {
  Attributes:
    length
    width
  Methods:
    getLength() { return length }
    getWidth()  { return width }
    draw()      { ... }
}
```

4. How would you implement the concept of class and method in a non-object-oriented programming language such as COBOL, Pascal or C?
5. Define using the following structure a class definition for cars. A car generally has abilities to start, move forward, move backward,

stop and off. A car can also return to its relative location. The starting location is a value 0.

```
class Car {
  Attributes:
    ...
  Methods:
    ...
}
```

6. Distinguish between a client and a server.
7. A client communicates with a server by sending a _____ to the server. The _____ is a call on a _____ of the server.