Chapter 3

Objects and Classes

In object-oriented programming, the principal modeling element is an *object*. An object is a distinct entity with a current state and a well-defined bevaviour. Objects are created from classes; thus, a *class* is a template for creating objects of the same type. This chapter explains how to write classes and how to create objects from these classes. It uses the concept of an account in a banking application to show how classes are written and how objects can be created and manipulated.

3.1 Understanding Classes

In object-oriented programming, a class is a template for creating objects of the same type. A class has a set of properties called *attributes*. Individual objects of the class have different values for the attributes. A class also defines a set of *behaviors* that are common to all members of the class. This section describes the attributes and behaviors of a class and shows how they are implemented in Java.

3.1.1 Representing State with Attributes

Attributes are properties of an object. They are used to store data about an object. The *state* of an object is the set of values for each of its attributes. Attributes are usually given noun-like names such as `number`, `balance`, `firstName`, `address`, etc. Attributes have types since the data they represent have types such as `int`, `double`, `String`, etc.

Consider an `Account` object in a banking application. This object will have attributes such as the following:

- `number`: a number which uniquely identifies the given `Account` object (an `int` value which serves as the *primary key*)

- `balance`: the current balance in the `Account` (a `double` value)

Attributes are usually given names that match the purpose they serve in an application. It is common to give names that closely match the real-world concepts being represented. Unintuitive or abbreviated names which are hard to decipher are generally not recommended since they are not considered good programming style.

The two attributes of an `Account` object can be declared as follows:

```
int number;                    // number attribute is of type int
double balance;                // balance attribute is of type double
```

3.1.2 Implementing Behaviors with Methods

An object has a set of behaviors. A *behavior* is a particular action or task that the object performs. A behavior depends on the current state of the object and will often result in the modification of that state. Behaviors are implemented by methods. A *method* is like a *function* of procedural programming languages and causes a certain action or operation to take place. A method usually has a verb-like name since it represents a particular task being done.

A method can be regarded as a *service* provided by an object. The object providing the service can be referred to as a *service provider* or *server object*. The object requesting the service can be referred to as a *service consumer* or a *client object*. A service can be requested by calling or invoking the corresponding method on the server object, sending any arguments that are required to provide the service.

Five behaviors of an `Account` object are listed in Table 3.1.

Behavior	*Description*
Deposit	Deposits a certain amount of money to an account.
Withdraw	Withdraws a certain amount of money from an account.
Set Number	Changes the account number of an account.
Get Balance	Finds out what is the balance in an account.
To String	Gives a string representation of the information available on an account.

Table 3.1: Five Behaviors of an `Account` Object

The code below shows how each of the behaviors listed in Table 3.1 can be implemented as a method in Java.

```
public void deposit(double amount) {        // Deposit behavior
   balance = balance + amount;
}

public void withdraw(double amount) {       // Withdraw behavior
   if (balance >= amount)
      balance = balance - amount;
}

public int setNumber(int n) {               // Set Number behavior
   number = n;
}

public double getBalance() {                // Get Balance behavior
   return balance;
}

public String toString() {                  // To String behavior
   String s;

   s = "Number: " + number + " Balance: " + balance;
   return s;
}
```

As can be observed from the code, the methods of an `Account` object can access the attributes of the object and can read and/or update the values of these attributes depending on their specific behaviors.

3.1.3 Creating a Class

A *class* is something like a template from which "real" objects are created. Thus, it is not really something tangible. An analogy is the pans that are used for baking. If it is shaped a certain way, all the cakes take on the shape of the pan. Thus, all the objects of the same class have the same set of attributes and the same set of behaviors. It is the combination of attribute values of an object which gives each object its state and differentiates objects from each other. It is not possible for an object to "leave out" an attribute of the class (though it is possible to not assign a value to the attribute in question). An object of a class is also referred to as an *instance* of the class.

In Java, the *class* structure combines the attributes and behaviors of an object into one unit. So, the attributes and methods described above can be combined together in an `Account` class as follows:

```
public class Account
{
   // declaration of attributes

   int number;
   double balance;

   // declaration and definition of methods

   public void deposit(double amount) {
      balance = balance + amount;
   }

   public void withdraw(double amount) {
      if (balance >= amount)
         balance = balance - amount;
   }

   public int setNumber(int n) {
      number = n;
   }

   public double getBalance() {
      return balance;
   }

   public String toString() {
      String s;

      s = "Number: " + number + " Balance: " + balance;
      return s;
   }
}
```

The keyword `public` used in the code above is referred to as an *access modifier*. Access modifiers are used to restrict access to the features of an object; they are discussed in more detail in Chapter 4. The `public` access modifier is used before a class declaration to specify that the class as a whole can be accessed by objects of any other class. This implies that objects of other classes can create and manipulate objects of the class whenever they require. The `public` access modifier is used before a method declaration to specify that the method can be invoked by an object of any other class.

As mentioned in Chapter 2, the file containing the source code for a class must have the same name as the name of the class, with the keyword `.java` appended. Thus, the file containing the `Account` class must be called `Account.java`. When `Account.java` is compiled, the `Account.class` file is produced if there are no compilation errors. If there are compilation errors, these must be fixed before the `Account.class` file is produced. Note that the `Account.class` file cannot be executed on its own since it does not contain a `main()` method.

It should be noted that the development of an object-oriented program is usually an incremental process. It is not necessary to write the code for all the methods of a class before compiling and testing the source code. A class can actually compile successfully with no attributes and no methods! So, if you are now starting out with object-oriented programming, you will find it very useful to write the code for methods incrementally as your understanding of the problem grows.

3.2 UML Notation for a Class

The UML notation for a class is a diagram consisting of three components. The top component is reserved for the name of the class. The middle component is used for the attributes of the class (with type specifiers if necessary). The bottom component is used to list the methods of the class. Figure 3.1 is a UML diagram of the `Account` class containing the attributes and methods discussed so far in this chapter.

The attribute component or the method component of a class may be left empty if the list of attributes or methods is not of interest at the moment. In addition, a single rectangle containing only the name of the class can be used to represent a class if there is no interest at the moment on attributes and methods. These variations of the UML diagram of the `Account` class are shown in Figure 3.2. The four types of UML diagrams shown in Figure 3.1 and Figure 3.2 are used throughout the book to represent a class.

It should be noted that the UML supports the specification of primitive data types as well as user-defined types, just like in Java and other object-oriented programming languages. Each primitive type is named as a complete word in English, beginning with an upper case letter; for example, `Integer`, `Double`, `Boolean`, etc. The type of an element such as an attribute, a parameter name, or a return value of a method is specified *after* the name of the element. Types can be omitted from a UML diagram in the interest of space or if the information provided does not add significant value to the diagram.

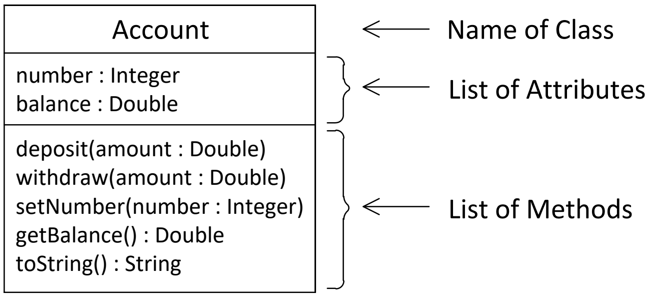

Figure 3.1: UML Diagram of `Account` Class

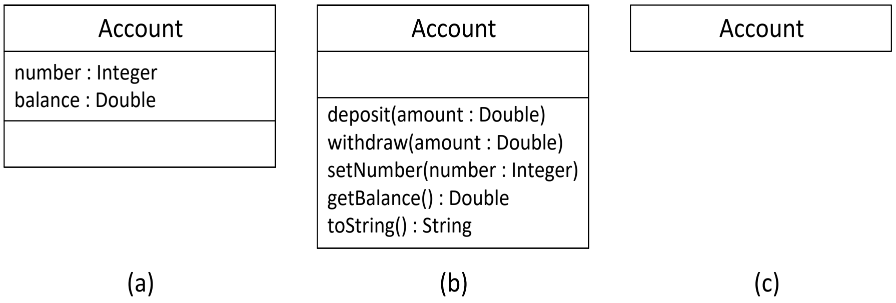

Figure 3.2: Variations of UML Diagram of `Account` Class (a) Methods are omitted (b) Attributes are omitted (c) Both methods and attributes are omitted

3.3 Creating and Manipulating Instances of a Class

This section shows how to create and manipulate instances of a class. In particular, it shows how to create `Account` objects and how to invoke methods on the `Account` objects. It uses various diagrams to show how client code modifies the state of an object in memory.

3.3.1 Creating and Manipulating Objects

In order to use the services of an object, an instance of the class must first be created. This is referred to as *instantiating* the class. An instance or object of the class can be created using the `new` keyword, followed by the name of the class, followed by a pair of parentheses. For example, to create an instance of the `Account` class, the following code can be used:

```
new Account();
```

This statement creates an `Account` object in memory. However, the object is lost immediately since there is no way to refer to the object just created. Thus, there is need for a variable which can refer to the object in memory. This variable is called an *object variable* and it holds a reference to an actual object in memory. (One can think of an object variable as containing the memory address of the created object.)

An object variable must be declared to be of the same type as the object to which it will refer. So, if an object variable is required to refer to an `Account` object, it must be declared to be of type `Account`:

```
Account a;            // a is an object variable of type Account
```

This statement declares `a` to be an object variable that will refer to a specific `Account` object. While the program is executing, `a` can refer to different `Account` objects. However, at any point in time, `a` can refer to at most one `Account` object. If `a` does not currently refer to a specific `Account` object, it should be set to `null` as follows:

```
a = null;
```

To assign a newly created `Account` object to `a`, the following code should be used:

```
a = new Account();
```

Once an object variable contains a reference to an instance of a class, services can be requested from that instance using method invocations on the object variable. Services are requested by specifying the name of the object variable, followed by a dot, followed by the name of the method, followed by a list of arguments (if any). For example, a client object can request the following services from the `Account` object, `a`:

Line 1:
```
a.setNumber(10);      // give the newly created account a number
```

Line 2:
```
a.deposit(1000.00);   // deposit $1000.00 to the account
```

Line 3:

```
System.out.println(a.toString());
                      // display contents of the account
```

Note that arguments must be supplied by the client object in Lines 1 and 2. The `toString()` method does not have any parameters so the argument list is empty.

Requesting a service from `a` when it does not currently refer to any object (i.e., `a` is `null`) results in a serious programming error. The compiler is sometimes able to detect this situation before it occurs. However, if it occurs at run-time, Java generates a `NullPointerException` and halts the program. Thus, it is important to ensure that object variables are not `null` (i.e., they refer to valid objects in memory) before requesting a service.

Throughout the book, whenever an object variable is used in the text, it refers to either the actual declared variable (e.g., `a` above) or to the object referred to by the object variable. The meaning will be clear from the context in which the object variable is used.

3.3.2 A Client Class

The code given in the previous sub-section is referred to as client code since it uses the services of the `Account` class. The client code can be written in another class, `BankApplication`. The code for the `BankApplication` class is given below:

```
public class BankApplication
{
   public static void main(String[] args) {

      Account a;      // declare object variable

      a = new Account();
                      // let a refer to object created

      a.setNumber(10);
                      // give the newly created account a number

      a.deposit(1000.00);
                      // deposit $1000.00 to the account

      System.out.println(a.toString());
                      // display contents of the account
   }
}
```

The `BankApplication` class must be saved in the file `BankApplication.java`. After compilation, the `BankApplication.class` file is produced. The `BankApplication` class *uses* the `Account` class since it creates an instance of the `Account` class and then proceeds to request services (invoke methods) from the instance. Thus, we now have a complete program where the functionality is achieved through the collaboration of two classes, `BankApplication` and `Account`. The program is available for download at the book Web site.

When the program is run, the following output is produced (generated from the `toString()` method of `a`):

```
Number: 10 Balance: 1000.0
```

3.3.3 Objects in Memory

The state of memory before the `BankApplication` program terminates is shown in Figure 3.3 (note that this is a variation of an actual UML diagram). The diagram shows that the object variable `a` refers to an instance of `Account` where the `number` attribute has the value 10 and the `balance` attribute has the value $1000.00.

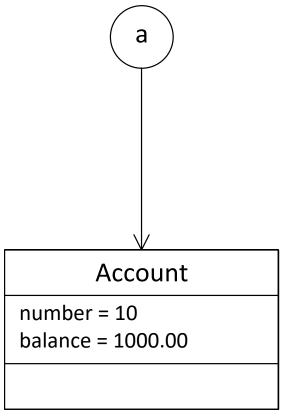

Figure 3.3: An Object Variable Referring to an Instance of `Account`

Suppose that the client code in `BankApplication` is extended to include code which creates another `Account` object and then proceeds to invoke methods on this object:

```
Account a2;                 // declare another object variable, a2

a2 = new Account();         // let a2 refer to object created
a2.setNumber(20);           // give the newly created account a number
a2.deposit(2000.00);        // deposit $2000.00 to the account
System.out.println(a2.toString());
                            // display contents of the account
```

When the program is run, the following output is produced:

```
      Number: 10 Balance: 1000.0
      Number: 20 Balance: 2000.0
```

The situation in memory before the program terminates is as follows. There are now two object variables, `a` and `a2`. `a` refers to one `Account` object in memory where the `number` attribute is 10 and the `balance` attribute is $1000.00. `a2` refers to another `Account` object in memory where the `number` attribute is 20 and the `balance` attribute is $2000.00. The state of memory is shown in Figure 3.4.

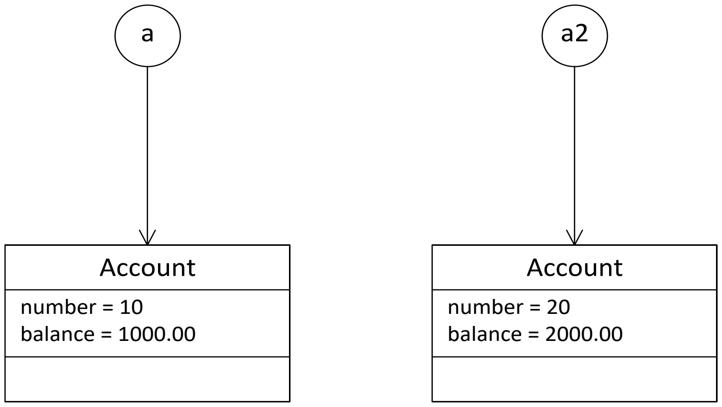

Figure 3.4: Two Object Variables Referring to Two `Account` Instances

So, there are two physical objects in memory, each one having its own state as determined by the values of its attributes. Each object will act independently of the other when services are requested of them. For example, consider the following code in a client:

```
a.withdraw(500.00);            // withdraw $500.00 from account #10
a2.withdraw(500.00);           // withdraw $500.00 from account #20
System.out.println(a.toString());
                               // display contents of account #10
```

```
System.out.println(a2.toString());
                              // display contents of account #20
```

The output generated from the above statements is as follows:

```
Number: 10 Balance: 500.0
Number: 20 Balance: 1500.0
```

When the `withdraw()` method is invoked on `a`, the withdrawal is successful and its `balance` is updated to $500.00 since its initial `balance` was $1000.00. However, when the `withdraw()` method is invoked on *a2*, its `balance` is updated to $1500.00 since its initial `balance` was $2000.00. Thus, even though the same `withdraw()` request was made of the two `Account` objects, the end result was different since the state of the objects was different. The state of the `Account` objects after the withdrawal is shown in Figure 3.5.

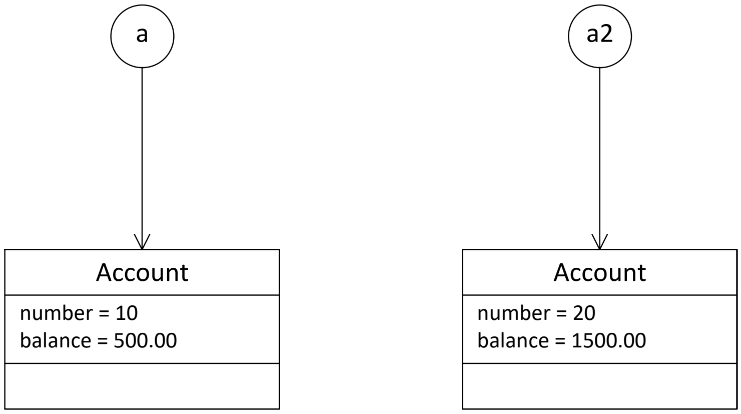

Figure 3.5: `Account` Objects after Withdrawal of $500.00

The behavior of individual objects in response to the same `withdraw()` request can be different as well. Consider the following code, given the state of the `Account` objects shown in Figure 3.5:

```
a.withdraw(600.00);           // withdraw $600.00 from account #10
a2.withdraw(600.00);          // withdraw $600.00 from account #20
System.out.println(a.toString());
                              // display contents of account #10
System.out.println(a2.toString());
                              // display contents of account #20
```

The output generated from the above statements is as follows:

```
Number: 10 Balance: 500.0
Number: 20 Balance: 900.0
```

The state of memory after the code is executed in shown in Figure 3.6.

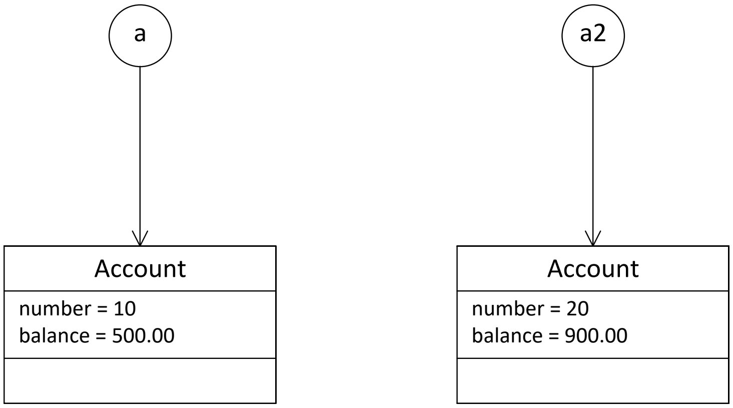

Figure 3.6: `Account` Objects after Withdrawal of Additional $600.00

Notice that the `balance` of `a` is unchanged. This is because the withdrawal amount is greater than the `balance` so the withdrawal did not take place. However, withdrawal of the $600.00 from `a2` is successful and its `balance` is updated to $900.00. This shows that different objects of the same class can behave differently when asked to do the same thing for a client since behaviors are governed by the state of an object. This is a key concept in object-oriented programming.

3.3.4 Assigning One Object Variable to Another

Consider what happens when we assign one object variable to another. For example,

```
a2 = a;
```

Assigning `a` to `a2` causes `a2` to have the same value as `a`. Thus, `a2` will now refer to the same object in memory as `a`. This is shown in Figure 3.7. It is important to note that assigning `a` to `a2` does not create a copy of the object referred to by `a`; only the object variable is copied.

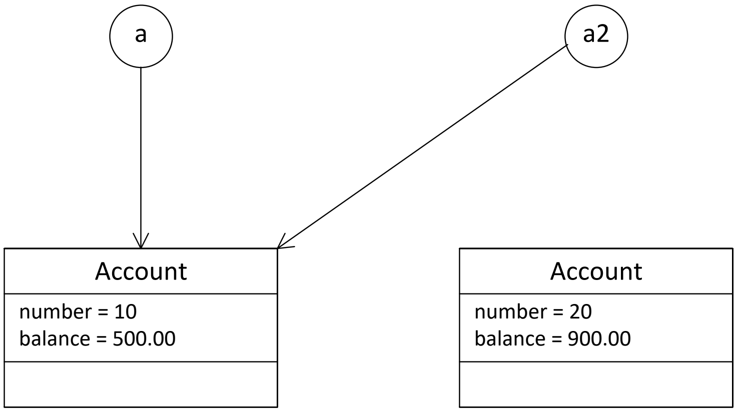

Figure 3.7: Both Object Variables Refer to the Same Object

If a method is invoked on either `a` or `a2`, the effect is the same since the same object in memory is being manipulated.

Eventually, the object formerly referred to by `a2` will be removed from the memory space of the application (by something known as the Java *garbage collector*) if there is no other object variable that refers to it.

3.4 Constructor Methods

A *constructor method* is a method that is used to initialize the attributes of an object after it has been created with the `new` keyword. The following is a constructor method for the `Account` class:

```
public Account(int n, double b) {
   number = n;
   balance = b;
}
```

Constructor methods always carry the same name as the class for which they are written (and must match exactly in terms of upper case and lower case characters). Thus, a constructor method for the `Account` class must have the name `Account`. Note that a constructor method does not have a return type. Constructor methods are the only kind of methods which do not have return types since they cannot be called like other methods.

The only time a constructor method comes into action is when an instance of a class is created with the `new` keyword. For example, after an `Account` instance is created, the constructor above sets its `number` and `balance` attributes to the arguments that were supplied when the constructor was called with the `new` keyword:

```
Account a = new Account(10, 1000.00);
```

This statement causes the value 10 to be assigned to the `number` attribute and the value 1000.00 to be assigned to the `balance` attribute of the newly created `Account` object.

Since a constructor method is just like a normal method (except that it does not specify a return type) it may take zero, one, or more arguments. The constructor above takes two arguments. If a constructor takes no arguments, it is called a *no-argument* constructor.

It is not necessary for a constructor method to have parameters for all the attributes of a class. For example, the following constructor for `Account` only accepts a parameter for the `number` attribute and sets the `balance` attribute to zero:

```
public Account(int n) {
   number = n;
   balance = 0.0;
}
```

In general, a constructor method should have parameters for attributes that a client can reasonably be expected to send when an instance is created. The other attributes are set to meaningful default values (like setting `balance` to zero in the example above).

A no-argument constructor enables clients to create an instance of a class without supplying any arguments. For example, consider the following no-argument constructor for the `Account` class:

```
public Account() {
   number = 0;
   balance = 0.0;
}
```

This constructor sets the `number` and `balance` attributes of an `Account` object to default values. When writing a no-argument constructor, the requirements of the application should be carefully considered. For example, in a banking application it probably makes no sense to have an `Account` object that does not have meaningful values for its `number` and `balance` attributes.

Consider the following code which was used in the previous section to create an instance of `Account`:

```
Account a;
a = new Account();
```

`Account()` is really a no-argument constructor that is used to initialize an `Account` object after it has been created by the `new` keyword. However, in the previous section, the `Account` class did not have any constructor methods. So, where did the no-argument constructor come from?

If a class contains no constructors, Java provides a *default no-argument constructor*. This constructor sets all the instance variables to default values which are appropriate for their type. So, numeric attributes (e.g. `double`, `int`, `long`, etc.) are set to zero, `boolean` attributes are set to `false`, character attributes are set to '*u0000*' (the Unicode representation of a `null` character—which sometimes prints incorrectly as a space), and object references are set to `null`. Thus, the `Account()` constructor is a default no-argument constructor supplied by Java. Note that Java will not supply the default no-argument constructor if one or more constructors are written for a class.

In specifying a parameter for a constructor method, programmers are often careful to use a name that does not conflict with name of the attribute being set. This results in parameter names that are sometimes quite meaningless. For example, the parameter `n` in the constructor method above is rather meaningless. To improve on this situation, one might consider using the name of the attribute as the parameter name. For example:

```
public Account(int number, double balance) {
   number = number;
   balance = balance;
}
```

This improves the readability of the parameter name; unfortunately, it results in a run-time error that often goes undetected. Since `number` and `balance` were declared in the parameter list, the reference to `number` and `balance` in the body of the method is actually a reference to the parameters, not the attributes of `Account`. This situation can be fixed by using the `this` keyword followed by a dot ('. ') as follows:

```
public Account(int number, double balance) {
   this.number = number;
   this.balance = balance;
}
```

The `this` keyword refers to the object of the class that contains the method that is currently being executed. Thus, `this`, followed by a dot, followed by the name of an attribute is a reference to an attribute of the current object. The use of `this` followed by a dot distinguishes the attribute name from the parameter name. In general, the `this` keyword can be used to distinguish the attributes of an object from the other variables declared in a method. This can be done in a constructor method as well as in the other methods of a class.

3.5 Improving the Client Class

The `BankApplication` class can be re-written to take advantage of the new constructor. In the following example, the `main()` method of `BankApplication` creates three `Account` objects and initializes them using the constructor. It then invokes methods on each object.

```
public class BankApplication
{
   public static void main(String[] args) {
      Account a1 = new Account(10, 1000.00);
      Account a2 = new Account(20, 2000.00);
      Account a3 = new Account(30, 3000.00);

      System.out.println (a1.getBalance());
      System.out.println (a2.toString());
      System.out.println (a3.getNumber());
   }
}
```

Since the constructor sets the account number to the value supplied as an argument, a `setNumber()` method is no longer required in the `Account` class. Indeed, having a `setNumber()` method is potentially dangerous since it is now possible for a client to modify `number`, the unique identifier of an `Account` object. So, the `setNumber()` method is removed from the `Account` class.

Instead of using three separate object variables in the code above, an array can be used to store the three `Account` objects. First, an array variable of type `Account` must be declared:

```
Account[] accounts;
```

The array is then created and assigned to the array variable as follows:

```
accounts = new Account[3];
```

Each `Account` object is created and initialized using the `new` keyword in conjunction with the constructor. The newly created `Account` object is then assigned to a specific location of the array using a subscript. The code below shows how the array is created and manipulated.

```
public class BankApplication
{
   public static void main(String[] args) {

      Account[] accounts;     // declare array variable

      accounts = new Account[3];
                              // create array

      accounts[0] = new Account(10, 1000.00);
                              // store account #10 in location 0

      accounts[1] = new Account(20, 2000.00);
                              // store account #20 in location 1

      accounts[2] = new Account(30, 3000.00);
                              // store account #30 in location 2

      for (int i=0; i<accounts.length; i++)
                              // display contents of each account

         System.out.println (accounts[i].toString());
   }
}
```

It should be noted that when the `accounts` array is created, the object variables are set to the default value of `null` since there are currently no `Account` objects in the array. This situation is depicted in Figure 3.8. Note that the *earth* (*ground*) symbol is used to indicate that an object reference is `null`.

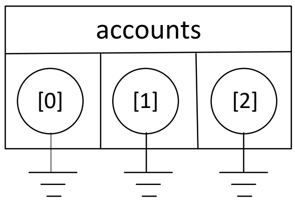

Figure 3.8: `accounts` Array Does Not Currently Refer to `Account` Objects

When an `Account` object is created, it is assigned to one of the array locations (just as if it were assigned to an ordinary object variable). To refer to a specific object variable in the array, the name of the array must be used, followed by the location of the `Account` object in the array. For example,

```
accounts[2]
```

Figure 3.9 depicts the `accounts` array after three instances of `Account` have been created and assigned to locations in the array:

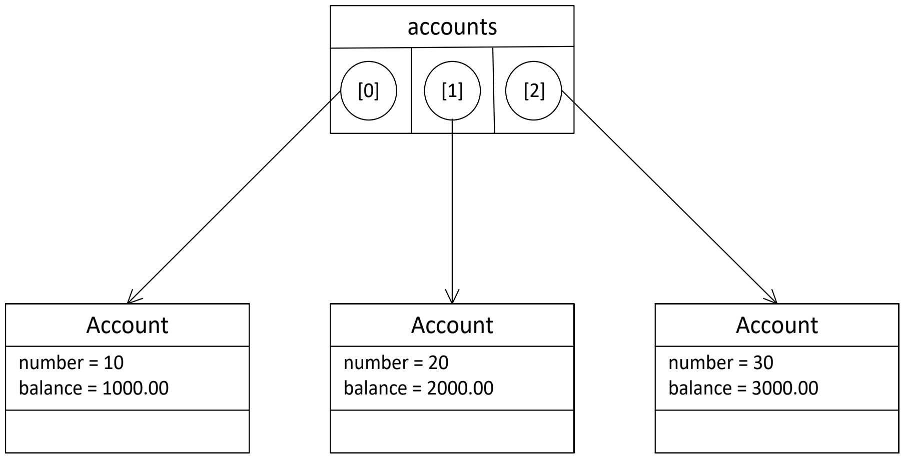

Figure 3.9: `accounts` Array Refers to Three `Account` Objects

Once the `accounts` array has been populated with references to actual `Account` objects, methods can be invoked on individual objects by using the object reference, followed by a dot, followed by the method name, followed by a list of arguments (if any). For example,

```
accounts[2].toString();
```

The `for` loop in the `main()` method of `BankApplication` uses an index variable `i` to traverse through each object variable in the array.

3.6 Method Signature and Method Overloading

Given the declaration of a method, the *method signature* consists of the name of the method, and the number, types, and order of its parameters. Consider the `deposit()` method of the `Account` class:

```
public void deposit(double amount) {
   balance = balance + amount;
}
```

The method signature of the `deposit()` method (underlined in the code above) is:

```
deposit(double amount)
```

Note that this method signature is identical to the following method signature since only the name of the dummy parameter has changed:

```
deposit(double value)
```

In Java, it is possible to define two or more methods in the same class with the same name but with different method signatures. When this is done, the methods are said to be *overloaded*. The process of writing methods with the same name is called *method overloading*. As an example, consider the `withdraw()` method of the `Account` class which is overloaded as follows:

```
// Version 1
public void withdraw(double amount) {
   balance = balance - amount;
}

// Version 2
public void withdraw(double amount, double fee) {
   balance = balance – amount - fee;
}
```

Suppose `a` is an instance of `Account`. The overloaded methods can be called as follows:

Line 1:
```
a.withdraw(100.00);
```

Line 2:
```
a.withdraw(200.00, 1.00);
```

When the `withdraw()` method in Line 1 is called, the `withdraw()` method that is used is the one with the single parameter (Version 1). When the `withdraw()` method in Line 2 is called, the `withdraw()` method that is used is the one with the two parameters (Version 2). It is easy for the compiler to resolve overloaded methods since the argument list in the method invocation is matched with the method signature in the declaration.

It is possible to write another version of the `withdraw()` method where the return type is changed from `void` to `int`. The code for this version is given below.

```
// Version 3
public int withdraw(double amount) {
                      // return type changed from void to int

   balance = balance - amount;
   return 0;
}
```

Notice that the signature of this method is the same as the `withdraw()` method in Version 1. A compile time error occurs indicating that the method in Version 1 is being re-declared. To fix the problem, one of the methods must be deleted. Alternatively, the signature of one of the methods must be changed so that the signatures of the two methods are different.

Constructor methods can also be overloaded. This makes it possible to create an instance of a class using different sets of initial values. Thus, the three constructors of the `Account` class which were previously described can all be used in the same class definition.

3.7 Class Variables and Class Methods

The attributes of a class can have different values for each instance of the class. Also, the methods of a class require an instance of the class to be present before a service can be requested (i.e., before invoking a method). A class can also have attributes that pertain to the class as a whole instead of individual instances. These are referred to as *class variables*. Similarly, a class can have methods that provide a service even if no instance of the class is available. These methods are referred to as *class methods*. This section discusses class variables and class methods.

3.7.1 Class Variables

Variables such as `number` and `balance` in the `Account` class are called *instance variables* since each instance of the class has its own set of values for these variables. If there are 100 instances of the `Account` class, there would be 100 sets of instance variables, each set containing its own values for the variables.

A *class variable* is a variable that is shared among all the instances of a class. Suppose the `Account` class needs a variable, `generator`, which is to be shared among all the instances of the `Account` class. `generator` is a class variable and is declared as follows:

```
static int generator;
```

The keyword `static` is used to distinguish a class variable from an instance variable. If there are 100 instances of the `Account` class, there is only one copy of `generator`, so there is only one value for the variable. If this value is modified by an object from the `Account` class, the changed value is immediately available to all the other objects of the `Account` class.

It is common to give a class variable an initial value when it is being declared. For example,

```
static int generator = 10000;
```

Class variables are useful for sharing data among instances of a class. For example, suppose that we wish to generate `Account` numbers automatically starting from 10000. We would like the `number` of each `Account` to be 10 higher than the `number` previously generated. We can share the `generator` among all instances of `Account` using a class variable. Whenever an `Account` instance is created, the `number` assigned is the value of the `generator`. The `generator` is then updated by 10 so the next instance will be assigned the incremented `generator` value. An example of a constructor method of `Account` which uses the `generator` class variable is given below:

```
public Account(double balance) {
            // NB: number is not a parameter

   number = generator;
            // account number assigned the value of generator

   this.balance = balance;
   generator = generator + 10;
            // prepare generator for next instance

}
```

3.7.2 Class Methods

All the methods described in the previous sections of this chapter can be called *instance methods* since they require an instance of a class to be present before they can be invoked. For example, the `deposit()` method of the `Account` class can only be invoked on an instance of `Account`:

```
Account a;
a = new Account(1000);
a.deposit(2000.00);
```

As previously mentioned, if `a` is not a valid reference to an `Account` instance, a run-time error occurs.

Unlike instance methods, *class methods* can be invoked without an instance of the class being present. A class method is declared just like an instance method except that the keyword `static` is used to indicate that it is a class method. For example, the `setGenerator()` method below is a class method:

```
public static void setGenerator (int newValue) {
   generator = newValue;
}
```

The `setGenerator()` method allows resetting of the `generator` value to a new starting value. Since it is a class method, it can be called without an instance of the class present.

In order to call a class method, you must specify the name of the class, followed by a dot, followed by the name of the class method and the argument list (if any). For example, the `setGenerator()` class method can be called as follows:

```
Account.setGenerator(20000);
```

A class method cannot refer to instance variables. Since it can be invoked without an instance present, no assumptions can be made about any instance of the class (and its associated instance variables). However, as can be seen from the `setGenerator()` method, a class method can refer to class variables. It can also refer to variables declared locally in the class method (see next section).

The client class, `BankApplication`, has a `main()` method which is declared with the `static` keyword. This implies that the `main()` method is a class method. Notice that `main()` only refers to variables declared locally in `main()`, e.g., `a1`, `a2`, etc.

3.8 Scope of Variables

An instance variable stores the value of an attribute of an object. In the `Account` class, `balance` is an instance variable. A *local variable* is a variable declared inside a method. For example, in the `calculateInterest()` method of the `Account` class shown below, `interest` is a local variable.

```
public void calculateInterest(double interestRate) {

   double interest;   // local variable declared inside method

   interest = balance * (interestRate / 12.0);

   // other processing
}
```

An instance variable of an object maintains its value as long as the object is "alive". In Java, "alive" is interpreted to mean that at least one other object in the program is currently referring to the given object. Of course, a client can change that value during the course of execution of the program. However, the important point to note about the state of an instance variable is that whenever a client accesses an object, the state of an instance variable is exactly the same as when the last client left it.

On the other hand, a local variable declared inside a method comes into existence when a method is invoked and "dies" when the method terminates (i.e., its former value cannot be accessed). The next time the method is invoked (from the same or from some other client), the local variable comes into existence again and dies when the method terminates. So, the value of a local variable does not persist from execution to execution of a method.

A class variable is "alive" from the moment the class is referenced in the program and stays "alive" until the program terminates. Whenever a client (or the `this` object) accesses a class variable, the value of the class variable is the same as when the last client left it.

Exercises

1. A class has attributes and behaviours. How are these two aspects of a class implemented in Java?

2. Draw the UML diagram for an `Employee` class which has attributes such as `number`, `name`, and `salary`, and methods such as

`raiseSalary(percentage)`, which raises an employee's `salary` by a certain percentage.

3. Two `Employee` objects have the same value for the `salary` attribute. The `raiseSalary()` method is invoked on each of them with a `percentage` argument of 5.0. Will the `salary` attribute of both objects have the same value after the `raiseSalary()` method is invoked? State any assumptions you make.

4. Figure 3.7 shows the result of copying an object variable `a` to another object variable `a2`. This results in a *shallow copy* of `Account` #10 being made since `a2` simply refers to the same `Account` #10. Explain how you would create a *deep copy* of `Account` #10, i.e., create a separate object which has the same attributes as `Account` #10. Write a method `deepCopy()` which returns a deep copy of the `Account` object on which it is invoked.

5. Explain why a class variable should not be initialized in a constructor method.

6. The methods of a class can be invoked on any instance of the class. Suppose a class has two types of instances for which certain methods are not applicable to all instances. For example, suppose that the `Employee` class has certain methods that are apply to only male instances and some that apply to only female instances. Suggest a technique for accomplishing this functionality in an object-oriented program (even in a limited way).

7. An array has been declared as follows:

```
Employees[] employees = new Employee[100];
```

Determine if the following code will run successfully:

```
for (int i=0; i<employees.length; i++)
   System.out.println(employees[i].toString());
```

8. A fraction is a number represented by two integer values: one for the numerator and the other for the denominator. For example, the fraction 3/5 is represented by the numerator 3 and the denominator 5. A `Fraction` class is required for storing and manipulating fractions. It should have methods such as `addFraction()`, `substractFraction()`, `multiplyFraction()`, and `divideFraction()` (each of which takes two arguments of type

`Fraction` and returns a new `Fraction` object). It should also have a `toString()` method which returns a string representation of the fraction.

(a) Draw a UML diagram of the `Fraction` class.

(b) Write the code for the `Fraction` class.

(c) Write the code for a client of `Fraction` that inputs several fractions from the user and performs various operations on the fractions.

9. Suppose that the constructor of the `Fraction` class receives a denominator of 0. Since a fraction cannot contain a denominator of 0, suggest a technique for dealing with this situation.

10. The `withdraw()` method of the `Account` class only performs the withdrawal if there are sufficient funds in an account. Suppose that an `Account` object does not have enough funds to perform a withdrawal. How can the `Account` object notify its client about this?