MT201

Unit 7

More on classes

Course team

Developer: Herbert Shiu, Consultant

Designer: Dr Rex G Sharman, OUHK

Coordinator: Kelvin Lee, OUHK

Member: Dr Vanessa Ng, OUHK

External Course Assessor

Professor Jimmy Lee, Chinese University of Hong Kong

Production

ETPU Publishing Team

The Open University of Hong Kong
30 Good Shepherd Street
Ho Man Tin, Kowloon
Hong Kong

Contents

Introduction

Up to this point in the course, we have discussed how to analyse a problem and derive the classes to model the real-world objects involved in it. From *Unit 4* to *Unit 6* you learned how to implement the object behaviours with different program constructs and arrays.

In *Unit 3*, while we were discussing the ways to define object attributes and behaviours, we mentioned that it was preferable to mark all attributes `private`. All methods in the unit were declared `public`. Such an approach leads to two programming concepts, *information hiding* (or *data hiding*) and *encapsulation*, which enable us to develop more reliable and maintainable software. We discuss the two concepts in detail in this unit.

Another scenario you might encounter is that while you are developing a software application with the Java programming language, you find that an existing class fulfils most of the requirements for a candidate class. Then, you are tempted to copy part of the existing class definition to the new one. However, you have to thoroughly understand the existing class definition to use such a 'copy-and-paste' approach. Furthermore, it is error-prone to do so, especially for the classes written by other programmers. If a bug is found in the original class definition, it is necessary to modify not only the original class definition but also all classes obtained by the 'copy-and-paste' approach. Each object-oriented programming language features a useful and powerful facility called *inheritance* that enables programmers to reuse or extend the functionality of existing class definitions.

In *Unit 2*, we said that in the analysis phase of the software development cycle, it is preferable to determine whether some classes are the specific classes of another general class. The implication is that if we can write a program that works with the general class, it can work with all specific classes of the general class. Such an approach motivates programmers to write class definitions that target general classes, which leaves room for programmers to write specific classes to be used by that class definition in the future. Such capabilities are achievable by the features of *inheritance*.

Usually, some behaviours of the specific classes are different from those of the general class. For example, when an analogue stopwatch receives a message `start`, its *hands* start moving; a digital stopwatch changes the *digits* shown when it receives the same message `start`. Any object-oriented programming language, such as the Java programming language, enables us to define a method in the specific classes to override the corresponding one in the general class. This is known as method overriding. As a result, objects of different specific types can behave differently when they receive the same message. Such a feature is known as *polymorphism*.

The above paragraphs give you some idea of how object-oriented programming enables you to reuse existing class definitions, write better

code and hence improve your software development productivity. They are essential facilities that an object-oriented programming language provides and thus have brought about the trend of software development using the object-oriented paradigm.

This unit concentrates on discussing the aforementioned features of the Java programming language. In addition to the core sections, nine appendices are included to elaborate on the ideas introduced in the unit, so that you can have a better understanding of some aspects of the language of object-oriented programming. The materials in the appendices are extra material — you can read them when you want to learn more.

Objectives

At the end of *Unit 7*, you should be able to:

1 *Describe* and *apply* information hiding and encapsulation.

2 *Describe* and *apply* inheritance.

3 *Describe* and *apply* polymorphism.

4 *Develop* class constructors.

5 *Describe* and *apply* abstract classes and methods.

Accessing object attributes directly and by invoking methods — encapsulation

In *Unit 3*, you learned that a class was a template or blueprint of a group of objects for modelling real-world objects of the same type. In a class definition, attributes and methods correspond to the object properties and behaviours respectively. You also learned that it was usual and preferable to mark all attributes `private` so that they must be accessed indirectly via methods.

In the following two subsections, we discuss the circumstances of accessing object attributes directly and indirectly by calling or invoking methods. Through using different implementations of a class design, you will realize the benefits of accessing object attributes via methods.

Accessing object attributes directly

Suppose that a software application needs a `Time1` object to denote a moment in time. The corresponding `Time1` class could be designed as shown in Figure 7.1.

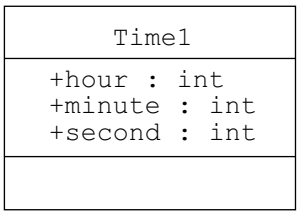


Figure 7.1 The design of the class Time1

The attributes `hour`, `minute` and `second` in the class design shown in Figure 7.1 are prefixed with a `+` character to indicate they are `public`. Declaring public attributes is usually a bad practice, and you are not advised to do so. Here, they are designed this way to let you see the problems. The corresponding class definition in the Java programming language is:

```
// Definition of class Time1
public class Time1 {
    // Attributes
    public int hour;
    public int minute;
    public int second;
}
```

All attributes of the class `Time1` are `public` so that they can be accessed directly by all objects. With such a class definition, you can create an object of the class `Time1` to present a moment in time. For

example, the following program segment creates a `Time1` object representing `12:30:00`.

```
Time1 t = new Time1();
t.hour = 12;
t.minute = 30;
t.second = 0;
```

The above approach leads to several problems.

1 Programmers who use the class have to thoroughly understand the internal representation of the attributes.

   Given the definition of the class `Time1`, programmers have to study it in detail to figure out the usage of each attribute. Furthermore, the programmers have to know the valid ranges of the attribute values, so that they will not assign invalid values to them.

   For example, for the class `Time1`, the valid range for the attribute `hour` is `0` to `23`, and the valid range for attributes `minute` and `second` is `0` to `59` for both. Therefore, it is the responsibility of the programmers who use the class to make sure that values assigned to the attributes are valid. Then, whenever a value is assigned to an object attribute, the programmer must provide a check beforehand or there is a potential risk that the software may not work as desired.

2 The attributes can be set arbitrarily without checking.

   A programmer who uses the above `Time1` class can create an object and set a value to the object attribute arbitrarily, even though the value to be assigned to an object attribute is invalid. For example, it is possible to have a statement segment like the following:

   ```
   Time1 t = new Time1();
   t.hour = -1;
   t.minute = -1;
   t.second = 60;
   ```

   The Java compiler software generates no compile time error because the statements are written according to the syntax of the Java programming language. However, it is obvious that the values assigned to the object attributes are invalid, which means that the `Time1` object is modelling a non-existing object or idea. The implication is that the software application will probably generate wrong or undesirable results.

   Therefore, a precaution is to ensure that all object attributes are properly set every time during the software execution. If the object attributes can be accessed directly, it is impossible to protect against such occurrences.

3 It is difficult to change the implementation of the class.

It is clearly shown in the class definition that the `Time1` class has three attributes: `hour`, `minute` and `second`. All statements that manipulate a `Time1` class object must use the exact attribute names, or compilation fails.

For whatever reason, if the attribute name is changed, it can be a nightmare to modify all statements that manipulate objects of the class `Time1`. It is not only time-consuming but also error-prone.

Accessing object attributes by invoking methods

From the previous subsection, you can see that marking object attributes non-`private` may cause three potential problems, and the reliability of the software is not guaranteed. To resolve the problems, it is good programming practice to mark attributes `private` and then to access them by invoking methods.

In the following sections, you will see how accessing object attributes by invoking methods can resolve the problems we mentioned.

Solving the internal representation knowledge problem (problem 1 above)

In order to prevent object attributes from being accessed directly, it is preferable to mark them `private` so that it is not possible to assign values to them with an assignment operation. Then, the class must provide corresponding methods for inquiring and updating the attribute values. The method for setting or updating an attribute value is known as a *mutator* method or simply a *setter* method. For inquiring an attribute value, another method should be provided to return the attribute value. Such a method is known as an *accessor* method or simply a *getter* method.

For example, for an attribute named `hour` of type `int`, the corresponding pair of getter/setter methods are:

```
private int hour;

public int getHour() {
   return hour;
}

public void setHour(int theHour) {
   hour = theHour;
}
```

Please notice that the methods are named according to the naming convention that every first letter of the words (other than the first word) in a method name is a capital letter. For example, if the name of an attribute is `annualInterest`, the getter/setter method names are

`getAnnualInterest()` and `setAnnualInterest()` respectively. A class definition can provide both setter and getter methods, or either one of them for an object attribute, so that an attribute can be made readable and writeable, readable only, or writeable only.

Another class `Time2` that can be used to represent a moment in time defines three attributes, and it therefore defines six corresponding getter/setter methods. The class design is shown in Figure 7.2.

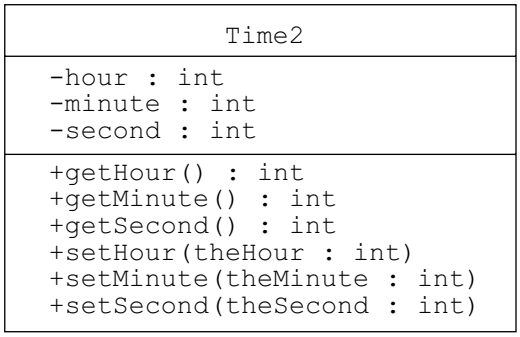


Figure 7.2 The design of the Time2 class with private attributes and public methods

For the diagram shown in Figure 7.2, the prefixes of the `-` and `+` characters indicate `private` members and `public` members of the class respectively (members of a class refer to methods and attributes). All attributes are marked `private` and all methods are marked `public` in the diagram.

Based on the above class design, the corresponding class definition is shown in Figure 7.3.

```
// Definition of class Time2
public class Time2 {
   // Attributes
    private int hour;      // The hour of time
    private int minute;    // The minute of time
    private int second;    // the second of time

    // Get the value of attribute hour
    public int getHour() {
       return hour;
   }

   // Get the value of attribute minute
   public int getMinute() {
      return minute;
   }

   // Get the value of attribute second
   public int getSecond() {
      return second;
   }

   // Set the value of attribute hour
```

```
    public void setHour(int theHour) {
       hour = theHour;
    }

    // Set the value of attribute minute
    public void setMinute(int theMinute) {
       minute = theMinute;
    }

    // Set the value of attribute second
    public void setSecond(int theSecond) {
       second = theSecond;
    }
 }
```

Figure 7.3 Time2.java

Then, instead of accessing the attribute directly, the object attributes must be accessed via method calls. For example:

```
Time2 t = new Time2();
t.setHour(12);
t.setMinute(30);
t.setSecond(0);
System.out.println(t.getHour() + ":" +
    t.getMinute() + ":" + t.getSecond());
```

Getter/setter methods enable the users of the objects to access object attributes indirectly, and they do not have to be concerned about their internal operations.

Solving the invalid data problem (problem 2 above)

Even though the object attributes can be accessed only by invoking methods, it is possible to assign invalid values to the object attributes of a `Time2` object, such as:

```
t.setHour(-1);
```

The reason is that the setter methods of the class do not validate the values to be assigned to the object attributes that are passed via setter method parameters. Therefore, it is necessary to enclose the assignment statement in an `if` statement, so that the attribute is updated only if the value supplied to the method via the parameter is valid. If a supplied value is checked and found invalid, it is preferable to notify the user, such as displaying a message on the screen.

A modified implementation, the definition of the `Time3` class, is shown in Figure 7.4.

```
// Definition of class Time3
public class Time3 {
   // Attributes
    private int hour;      // The hour of time
    private int minute;    // The minute of time
    private int second;    // the second of time

    // Get the value of attribute hour
    public int getHour() {
       return hour;
   }

   // Get the value of attribute minute
   public int getMinute() {
      return minute;
   }

   // Get the value of attribute second
   public int getSecond() {
      return second;
   }

   // Set the value of attribute hour
   public void setHour(int theHour) {
      if (0 <= theHour && theHour < 24) {
          hour = theHour;
       }
       else {
          System.out.println(
                "Invalid hour. Object attribute kept unchanged.");
      }
   }

   // Set the value of attribute minute
   public void setMinute(int theMinute) {
      if (0 <= theMinute && theMinute < 60) {
         minute = theMinute;
      }
      else {
         System.out.println(
                "Invalid minute. Object attribute kept unchanged.");
      }
   }

   // Set the value of attribute second
   public void setSecond(int theSecond) {
      if (0 <= theSecond && theSecond < 60) {
         second = theSecond;
      }
      else {
         System.out.println(
          "Invalid second. Object attribute kept unchanged.");
      }
   }
}
```

Figure 7.4 Time3.java

If an attempt is made to update an attribute, say `hour`, of a `Time3` object with an invalid value, an error message will be shown on the screen. For example, when the following statements create a `Time3` object and attempt to update the object attributes with invalid values

```
Time3 t = new Time3();
t.setHour(-1);
t.setMinute(-1);
t.setSecond(60);
```

the following output is shown on the screen:

```
Invalid hour. Object attribute kept unchanged.
Invalid minute. Object attribute kept unchanged.
Invalid second. Object attribute kept unchanged.
```

Such an approach ensures that the object attributes of the `Time3` object are kept unreachable from the objects that use it, which is known as *information hiding* (or *data hiding*). Information hiding can improve software reliability, and it is easier to pinpoint software bugs because it ensures the object attributes are valid throughout the software execution.

Solving the implementation change problem (problem 3 above)

Before we discuss how to solve the problem, let's make necessary changes to the class. The class design is further enhanced so that it defines a method `tick()` to advance the time by one second and a `findDifference()` method to find the difference between two times in seconds. Another method named `toString()` is defined; it returns the reference of a `String` object as a textual representation of the object. The design of the class, say `Time4`, is shown in Figure 7.5.

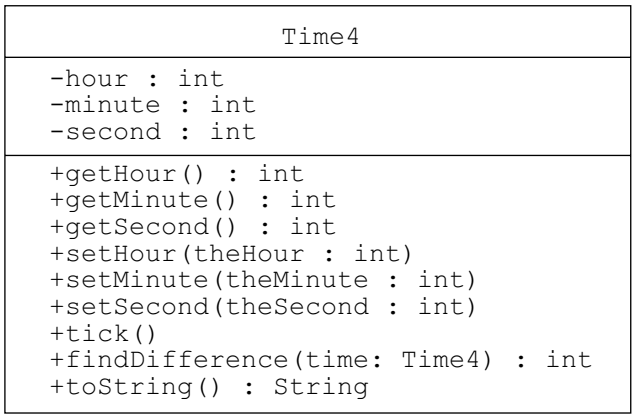


Figure 7.5 The design of the Time4 class with private attributes and public methods

The corresponding definition of the `Time4` class is shown in Figure 7.6.

```
// Definition of class Time4
public class Time4 {
    // Attributes
    private int hour;      // The hour of time
    private int minute;    // The minute of time
    private int second;    // the second of time

    // Get the value of attribute hour
    public int getHour() {
        return hour;
    }

    // Get the value of attribute minute
    public int getMinute() {
        return minute;
    }

    // Get the value of attribute second
    public int getSecond() {
        return second;
    }

    // Set the value of attribute hour
    public void setHour(int theHour) {
        if (0 <= theHour && theHour < 24) {
            hour = theHour;
        }
        else {
            System.out.println(
                "Invalid hour. Object attribute kept unchanged.");
        }
    }

    // Set the value of attribute minute
    public void setMinute(int theMinute) {
        if (0 <= theMinute && theMinute < 60) {
            minute = theMinute;
        }
        else {
            System.out.println(
                "Invalid minute. Object attribute kept unchanged.");
        }
    }

    // Set the value of attribute second
    public void setSecond(int theSecond) {
        if (0 <= theSecond && theSecond < 60) {
            second = theSecond;
        }
        else {
            System.out.println(
             "Invalid second. Object attribute kept unchanged.");
        }
    }

    // Advance the time by one second
```

```
    public void tick() {
        second++;
        if (second == 60) {
            second = 0;
            minute++;
        }
        if (minute == 60) {
            minute = 0;
            hour++;
        }
        if (hour == 24) {
            hour = 0;
        }
    }

    // Determine the difference between the given time
    // in seconds
    public int findDifference(Time4 time) {
        return
            (second - time.getSecond()) +
            (minute - time.getMinute()) * 60 +
            (hour - time.getHour()) * 3600;
    }

    // Show the time in the format hh:mm:ss on the screen
    public String toString() {
        return hour + ":" + minute + ":" + second;
    }
}
```

Figure 7.6 Time4.java

The `findDifference()` method accepts a parameter of type `Time4`, which means that it accepts a reference of a `Time4` object, and it can then send a message to the `Time4` object referred by the parameter `time`.

To test the `Time4` class, a driver program named `TestTime4` is written. The definition is given in Figure 7.7.

```
// Definition of class TestTime4
public class TestTime4 {
    // Main executive method
    public static void main(String args[]) {
        // Create and initialize a Time4 object
        Time4 t1 = new Time4();
        t1.setHour(Integer.parseInt(args[0]));
        t1.setMinute(Integer.parseInt(args[1]));
        t1.setSecond(Integer.parseInt(args[2]));
        // Show the Time4 object on the screen
        System.out.println("T1=" + t1.toString());

        // Create and initialize a Time4 object
        Time4 t2 = new Time4();
        t2.setHour(Integer.parseInt(args[3]));
```

```
        t2.setMinute(Integer.parseInt(args[4]));
        t2.setSecond(Integer.parseInt(args[5]));
        // Show the Time4 object on the screen
        System.out.println("T2=" + t2.toString());

        // Calculate and display the difference between the two times
        System.out.println("Difference=" + t1.findDifference(t2));
    }
}
```

Figure 7.7 TestTime4.java

The `TestTime4` class expects six program parameters that represent two time instances. The usage of the `TestTime3` class is:

`java TestTime4 <Hour1> <Minute1> <Second1> <Hour2> <Minute2> <Second2>`

For example, compiling the classes `Time4` and `TestTime4` and executing the `TestTime4` with the following command,

`java TestTime4 13 15 30 10 30 0`

the program finds the difference between the times `13:15:30` and `10:30:00` and the following output is shown on the screen:

```
T1=13:15:30
T2=10:30:0
Difference=9930
```

Changing internal implementation

Programs that manipulate `Time4` objects can access or call all their `public` members only, which are the methods of the class `Time4`. Therefore, provided that the ways to call the methods (the combinations of method names and the parameter lists) are kept unchanged, it is possible to change the way the methods are implemented in the class definition.

For example, instead of storing the values for `hour`, `minute` and `second` separately in three attributes, it is possible to store a single value as the number of seconds since midnight to represent a moment in time. Then, another class, `Time5`, is written and the definitions of the methods `tick()` and `findDifference()` are simply:

```
// Advance the time by one second
public void tick() {
    secondSinceMidNight = (secondSinceMidNight + 1) % 86400;
}

// Determine the difference between the given time
// in seconds
public int findDifference(Time5 time) {
  return secondSinceMidNight - time.secondSinceMidNight;
}
```

Sending a message `tick` to a `Time5` object, the `Time5` object advances the time it represents by one second. Therefore, the basic operation for a `tick` behaviour is:

```
secondSinceMidNight++;
```

However, if it is the last second of the day, `23:59:59`, the value of the attribute `secondSinceMidNight` is `86399` (`23 × 3600 + 59 × 60 + 59`), the time becomes `00:00:00` if the time is advanced by one second and the value of the attribute `secondSinceMidNight` becomes `86400`, but its value should be `0` to represent the time `00:00:00`. Therefore, it is necessary to handle such a situation. A possible way is to use an `if` statement to rectify the value of the attribute `secondSinceMidNight`, such as:

```
secondSinceMidNight++;
if (secondSinceMidNight >= 86400) {
    secondSinceMidNight = 0;
}
```

An even simpler way is to use the `%` operator to calculate the remainder of an integer division. If the value of the attribute `secondSinceMidNight` equals `86399`, the result of `(secondSinceMidNight + 1) % 86400` will be `0`. For other values of the attribute `secondSinceMidnight` less than `86399`, the value will be simply increased by one.

The above two methods are much simpler than the equivalent ones of the `Time4` class. However, the other getter/setter methods are slightly more complicated. The complete definition of the `Time5` class is provided in Figure 7.8.

```
// Definition of class Time5
public class Time5 {
    // Attributes
    private int secondSinceMidnight;

    // Get the value of attribute hour
    public int getHour() {
        return secondSinceMidnight / 3600;
    }

    // Get the value of attribute minute
    public int getMinute() {
        return (secondSinceMidnight % 3600) / 60;
    }

    // Get the value of attribute second
    public int getSecond() {
        return (secondSinceMidnight % 60);
    }

    // Set the value of attribute hour
    public void setHour(int theHour) {
```

```
        if (0 <= theHour && theHour < 24) {
            secondSinceMidnight =
                theHour * 3600 +
                (secondSinceMidnight % 3600);
        }
        else {
            System.out.println(
                "Invalid hour. Object attribute kept unchanged.");
        }
    }

    // Set the value of attribute minute
    public void setMinute(int theMinute) {
        if (0 <= theMinute && theMinute < 60) {
            secondSinceMidnight =
                (secondSinceMidnight / 3600 * 3600) +
                theMinute * 60 +
                (secondSinceMidnight % 60);
        }
        else {
            System.out.println(
                "Invalid minute. Object attribute kept unchanged.");
        }
    }

    // Set the value of attribute second
    public void setSecond(int theSecond) {
        if (0 <= theSecond && theSecond < 60) {
            secondSinceMidnight =
                secondSinceMidnight / 60 * 60 +
                theSecond;
        }
        else {
            System.out.println(
                "Invalid second. Object attribute kept unchanged.");
        }
    }

    // Advance the time by one second
    public void tick() {
        secondSinceMidnight = (secondSinceMidnight + 1) % 86400;
    }

    // Determine the difference between the given time
    // in seconds
    public int findDifference(Time5 time) {
        return secondSinceMidnight - time.secondSinceMidnight;
    }

    public String toString() {
        return getHour() + ":" + getMinute() + ":" + getSecond();
    }
}
```

Figure 7.8 Time5.java

In the `getMinute()` method, the expression `secondSinceMidnight % 3600` is used to find the number of seconds left after removing all whole hours and dividing it by `60` to get the number of minutes. In the `setMinute()` method, the expression after checking is equivalent to

```
secondSinceMidnight = getHour()*3600 + theMinute*60 + getSecond();
```

which should be easier to understand. Please verify that other getter/setter methods indeed perform the intended actions.

As the `Time5` class uses a single attribute `secondSinceMidnight` to indicate the number of seconds since midnight, the getter/setter methods are not as straightforward as those defined in the `Time4` class. Even though their implementations are different, both `Time4` and `Time5` classes have the same set of `public` members. As a result, it is possible to manipulate a `Time5` object in exactly the same way as using `Time4` objects. For example, another driver program, `TestTime5` that is similar to the `TestTime4` class (except that the objects to be created are `Time5` objects instead of `Time4` objects) is written to test `Time5` objects. The definition of the class `TestTime5` is not shown in the unit. Please refer to the CD-ROM or the course website for it.

The definitions of `Time4` and `Time5` classes illustrate another important idea. Based on the same class design, the classes `Time4` and `Time5` use different implementations, but the ways to use objects of classes `Time4` or `Time5` are the same. The reason is that they have the same set of `public` methods to be used by other classes, and all internal implementations (such as attributes) are hidden. Then, the users of the classes can manipulate an object of classes `Time4` or `Time5` in the same way and need not be concerned about their internal operations. In other words, the objects of both classes can accept the same set of messages, but their internal operations are hidden from the users of the classes. Such an approach is known as *encapsulation*.

Another benefit of encapsulation is that if a class is written with encapsulation in mind and there is a better implementation, it is possible to change the implementation. The modifications are localized in a single class, and no other class definition needs changes.

Information hiding and encapsulation summary

Information hiding and encapsulation concern the ways to design and implement classes in an object-oriented programming language like the Java programming language. Both can promote writing better and more reliable software.

Information hiding is hiding the data (attributes) of an object from the outside world, and all accesses and updates to them must be done by invoking methods so that an object can validate the values to be assigned to the object attributes. This prevents an object from possessing invalid object attribute values or states.

Encapsulation concerns the hiding of the internal implementation details so that the users of the object (or class) use an object by calling its methods only. Furthermore, it is not necessary for the user of the object to understand the detailed operations of each method but just to send messages and supply the necessary supplementary data to an object and it can perform the behaviour accordingly. As a result, provided that a behaviour of an object can produce the same result, it is up to the object (and hence the corresponding class) to define its own way of carrying out the behaviour.

While designing a class, the above two concepts — information hiding and encapsulation — are implemented if all mutable (or updateable) object attributes are marked `private` and their values can only be changed by calling the corresponding non-`private` methods of the object.

Self-test 7.1

A programmer develops a software application for bookkeeping of the monthly payroll records of all staff members in a company. After the analysis and design phases, it is determined that a class `PayrollRecord` is required with the following attributes:

- year — the year of the monthly payroll record
- month — the month of the monthly payroll record
- staff number — the unique staff number of the staff member
- salary — the salary for the month.

To assist the programmer, you are requested to perform the following tasks:

1 Choose a suitable type for each attribute of the class `PayrollRecord`. (If necessary, please refer to *Unit 3* for the discussion on choosing the types.)

2 Determine the validation condition for each attribute.

3 Write the complete definition of the class `PayrollRecord` with information hiding and encapsulation in mind.

The following reading is about how to choose suitable instance variables. It is related to information hiding.

Reading

King, section 10.2, pp. 391–95

Initializing objects during creation — constructors

Since *Unit 3*, you have been designing classes and implementing them using class definitions written in the Java programming language. For example, for the first class you came across, `TicketCounter`, we mentioned that the class definition was a template and we could use the keyword `new` to create an object of the class, such as:

```
new TicketCounter()
```

We said that the above statement created an object based on the definition of the class `TicketCounter`, and the reference of this newly created object was returned. Behind the scenes during the creation process of an object, a special method of the class known as a *constructor* is executed. In the following subsection, we discuss what constructors are and the entire creation process of an object.

Defining constructors

The constructor of a class can be treated as a special method that is executed during the process of creating an object. The format of the constructor for a class is:

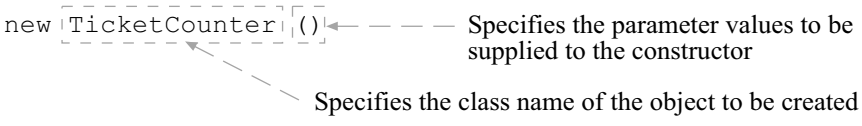
```
public Class-name(Parameter-list) {
    ......
}
```

Compared with the format for a usual method, a constructor has no return type and the name must match the class name. For example, the constructor for the class `TicketCounter` can be:

```
public TicketCounter() {
    reading = 0;
}
```

The single statement of the method body assigns a value of zero to the object attribute `reading`. Besides initializing the instance variables, constructors are usually used to set up the necessary resources that the object requires.

When an object is created using the keyword `new`, the pair of parentheses that follow the class name specifies the supplementary data to be supplied to the constructor. For example:



In other words, the parameter value list supplies data to the constructor via the constructor parameter list. For example, with respect to the above constructor, the following statement

```
new TicketCounter()
```

creates a `TicketCounter` object, and the empty parameter value matches the constructor with an empty parameter list and provides no supplementary data to the constructor. Therefore, the constructor without a parameter list is executed during the creation process.

The definition of the class `TicketCounter` should provide a constructor that looks like the following one:

```
public TicketCounter() {
    ......
}
```

You might wonder why the definitions of all classes that we have discussed so far define no constructor, but we can create objects of the classes with statements like:

```
new TicketCounter()
```

The reason is that if a class defines no constructor, a *default constructor* or a 'no-arg' constructor with an empty parameter list and an empty method body is implicitly added to it. For example, if the definition of the class `TicketCounter` defines no constructor, the following constructor is added to the class implicitly:

```
public TicketCounter() {
    // No statement
}
```

Therefore, we can use the following statement to create a `TicketCounter` object without defining a constructor ourselves:

```
new TicketCounter()
```

Constructors with parameters

A constructor can have a parameter list that enables you to provide supplementary data to it for proper initialization of the object. For example, it is possible to define a constructor for the `TicketCounter` class with a parameter of type `int`, and the value of type `int` supplied to the constructor can be used as the initial value of the attribute `reading`. The definition can be written as follows:

```
public TicketCounter(int theReading) {
    reading = theReading;
}
```

Then, the parameter value list to be used for creating a `TicketCounter` class object must match the parameter list, such as:

```
new TicketCounter(100)
```

The above statement creates an object of the class `TicketCounter`, and the value `100` of type `int` is supplied to the constructor via the parameter `theReading`.

If a class defines a constructor, no default constructor will be added implicitly. Unless the parameter list of the explicitly defined constructor is empty, you cannot create an object of the class using a `new` statement with an empty parameter value list. For example, if the `TicketCounter` class defines a constructor like the following,

```
public TicketCounter(int theReading) {
    reading = theReading;
}
```

the `TicketCounter` class contains a constructor that expects a parameter of type `int`. Then, the following statement,

```
new TicketCounter()
```

would cause a compile-time error because the compiler software cannot find a constructor with an empty parameter list. Later in this unit, we discuss a way that enables you to define two constructors so that you can create a `TicketCounter` object using either way.

The creation process

Creating an object with the keyword `new` involves a few steps. The following is a sequence of steps taken during the creation process:

1 Memory allocation. A software object occupies memory space in the Java Virtual Machine (JVM), and the JVM allocates the necessary memory for it first.

2 Implicit initialization. All instance (non-class or non-`static`) variables of the object are implicitly initialized. The initial values are determined according to the variable type as shown in Table 7.1, which are identical to the initial values of array elements after the array object is created.

3 Explicit initialization. If an instance variable is declared with initialization, it is initialized according to the initialization expression.

4 Execution of the constructor. The constructor of the object is executed. If the constructor accepts parameters, the values for the parameters are supplied to it for execution.

Table 7.1 Initial values of instance variables (object attributes) of different types

Variable type	Variable initial value
`byte`	`(byte) 0`
`short`	`(short) 0`
`char`	`(char) 0`
`int`	`0`
`long`	`0L`
`float`	`0.0F`
`double`	`0.0D`
`boolean`	`false`
All non-primitive types	`null`

For illustration, a class `TicketCounter` is modified to be `TicketCounter1` as shown in Figure 7.9.

```
// Definition of class TicketCounter1
public class TicketCounter1 {
   // Attribute
   private int reading = 1; // The reading of the counter

   // The constructor
   public TicketCounter1(int theReading) {
      reading = theReading;
   }

   // Increase the reading by one
   public void increase() {
      reading++;
   }

   // Increase the reading by the specified amount
   public void increaseByAmount(int amount) {
      reading += amount;
   }

   // Retrieve the value of the attribute reading
   public int getReading() {
      return reading;
   }
}
```

Figure 7.9 TicketCounter1.java

To create a `TicketCounter1` object, it is necessary to provide a value of type `int` via the parameter `theReading` because the constructor needs it. A possible statement to create the object is:

```
TicketCounter1 counter = new TicketCounter1(100);
```

Let's trace the steps in executing the above statement.

1 The variable `counter` is declared. As its type is non-primitive, a block of memory space that can store a reference to a `TicketCounter1` object is allocated and is denoted by the identifier `counter`.

2 The expression `new TicketCounter1(100)` is executed. First of all, a block of memory space for a `TicketCounter1` object is allocated. The size of the memory block is determined automatically by the JVM. The details are not our concern.

The `TicketCounter1` object is visualized as shown in Figure 7.10.

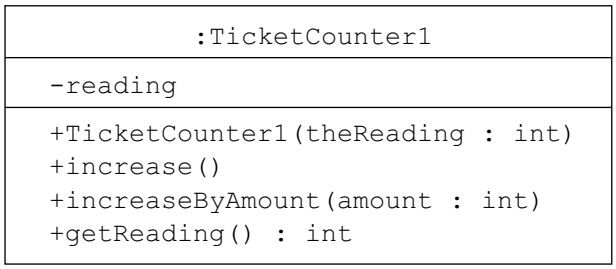


Figure 7.10 A memory block is allocated to a TicketCounter1 object in the JVM

3 The instance variable of the object is implicitly initialized with values according to the variable type shown in Table 7.1. For the `TicketCounter1` object, the attribute `reading` is of type `int` and it is therefore initialized to be `0`. Then, the object is visualized as shown in Figure 7.11.

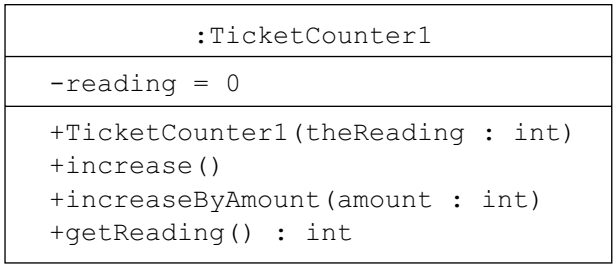


Figure 7.11 The TicketCounter1 object after implicit initialization

4 The declaration of the instance variable `reading` in the `TicketCounter1` class definition comes with an initialization.

```
private int reading = 1;
```
← initialization

Therefore, the instance variable `reading` is further updated to be `1`. The object is shown in Figure 7.12.

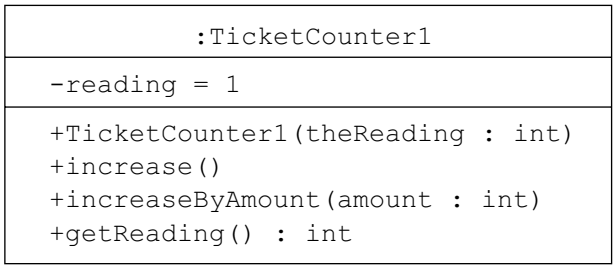


Figure 7.12 The TicketCounter1 object after explicit initialization

5 The constructor of the object is executed. As the parameter value list enclosed in the parentheses,

```
new TicketCounter1(100)
```

is a value of `100` of type `int`, the value of `100` is supplied to the constructor and the constructor parameter `theReading` is assigned a value of `100`.

When the constructor is executed, its statement,

```
reading = theReading;
```

assigns the value of the parameter `theReading` to the object instance variable `reading`. Therefore, the instance variable is further updated to be `100` as shown in Figure 7.13.

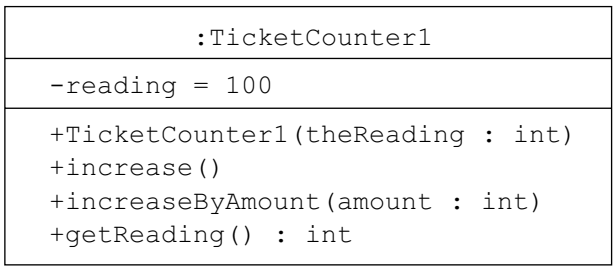


Figure 7.13 The TicketCounter1 object after the constructor is executed

The creation process of the `TicketCounter1` object is now complete.

6 Finally, the reference of the newly created `TicketCounter1` object is assigned to the variable `counter`. The scenario is visualized in Figure 7.14.

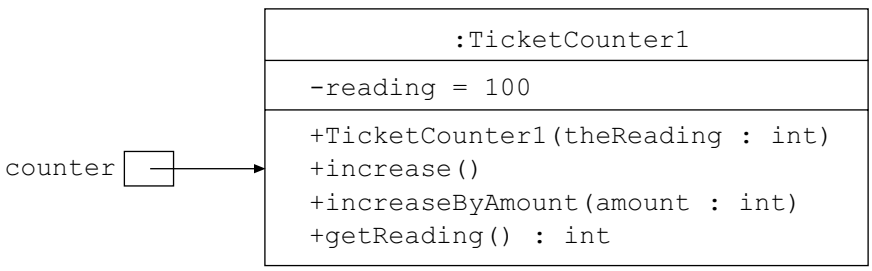


Figure 7.14 The final scenario of the TicketCounter1 object

In *Unit 3*, we said that you could define class members, including class variables and class methods, which are marked with the keyword `static`. The ways of allocating and initializing class variables are different from instance (non-`static`) variables. They are allocated and initialized when the class definition is loaded into the JVM. The JVM loads each class once, and the initialization of class variables is therefore only performed once; no objects of the class have been created. You can provide an initialization for a class variable to explicitly initialize it. Otherwise, it is implicitly initialized with an initial value according to its type as shown in Table 7.1.

For example, a class `TicketCounter2` is defined in Figure 7.15.

```
// Definition of class TicketCounter2
public class TicketCounter2 {
   // Attribute
   public static int limit = 1000;
   private int reading;  // The reading of the counter

   // The constructor
   public TicketCounter2(int theReading) {
      reading = theReading % limit;
   }

   // Increase the reading by one
   public void increase() {
      reading = (++reading) % limit;
   }

   // Increase the reading by the specified amount
   public void increaseByAmount(int amount) {
      reading = (reading + amount) % limit;
   }

   // Retrieve the value of the attribute reading
   public int getReading() {
      return reading;
   }
}
```

Figure 7.15 TicketCounter2.java

Compared with the class `TicketCounter1`, a `TicketCounter2` object supports the attribute `reading` ranging from `0` to `999` inclusive. The class defines a class variable `limit` that specifies the number of possible readings a `TicketCounter2` object supports. As all objects of the class have the same limit, that quantity is stored as a class variable so that it is shared among all objects of the class.

Whenever the JVM needs the definition of the class `TicketCounter2` for the first time, such as a declaration of variable of type `TicketCounter2`:

```
TicketCounter2 counter;
```

the JVM reads the class definition (`TicketCounter2.class`) probably from the computer hard disk. Then, all class members of the `TicketCounter2` class become available. The class methods can be executed, and the class variables are allocated in the memory and are initialized. With respect to the `TicketCounter2` class, the class variable `limit` is allocated and is initialized with a value of `1000`. You should notice that no `TicketCounter2` object is created for the time being.

Self-test 7.2

1 Review the definition of the class `IntegerStack5` discussed in *Unit 5*. Define a constructor for it, so that the initial size of its array object is determined by the constructor parameter of type `int`.

That is, the following expression

```
new IntegerStack5(100)
```

creates an `IntegerStack5` object with an array object of size `100` initially.

(Hint: The explicit initialization of the array variable is removed. A constructor looking like the following one is added to the class, and the constructor creates the array object.)

```
public IntegerStack5(int initialCapacity) {
   ......
}
```

2 In the section 'Example program — calculating the total price of items at the cashier's counter' of *Unit 5*, three classes, `Cashier`, `PurchaseOrder` and `Item` are defined. Define a constructor for the `PurchaseOrder` class that needs a supplementary datum of item total, and hence an array object of suitable size is created and used. Furthermore, create a constructor for the `Item` class so that the values of name, price and quantity are supplied to it via constructor parameters for initializing the object. As a consequence, the method `addItem()` of the class `PurchaseOrder` and `main()` method of `Cashier` are modified so that the constructors are called and are supplied with necessary parameter values.

In this section, we discussed the ways to define class constructors and the creation process. In a later section, we revisit them and have an in-depth discussion.

You can find some more examples about constructors in the following reading.

Reading

King, section 10.5, pp. 406–8

`Final` variables

Because a class variable (which is marked with keyword `static`) is shared among all objects of the class, if one of the objects accidentally changes its value, all other objects will get that value the next time they access it. Therefore, it is usually preferable to make the variable immutable or read-only if their values should not be changed, so that the objects can get the same value every time they access it. To make a variable immutable, we use the keyword `final`, such as:

```
private static final int limit = 1000;
```

To highlight an immutable variable, the naming convention of the Java programming language recommends that the name of an immutable variable be composed of capital letters only. If the variable name contains more than one word, the words are separated by underscore characters '_'. Therefore, the above class variable declaration is usually written as:

```
private final static int LIMIT = 1000;
```

Sometimes, the immutable class variable is marked `public` instead of `private`. As it is not possible to change the value of the immutable variable, it is safe to access it directly.

The content of an immutable non-primitive variable is a reference to an object. Such a reference cannot be changed, but the object referred by the variable can be changed. For example, the following statement declares an immutable array variable `numbers` and refers to an array object with five array elements:

```
final int[] numbers = new int[5];
```

Then, it is not possible to execute a statement to change the content to make it refer to another array object. For example, the following statement

```
numbers = new int[6];
```

would cause a compile-time error, because you intend to update the content of an immutable variable. However, it is possible to change the array elements of the array object that is referred by the variable `numbers`. Therefore, the statement

```
numbers[0] = 10;
```

is valid, and the first element of the array object referred by the variable `numbers` is updated to be `10`. Figure 7.16 can help you understand the idea better.

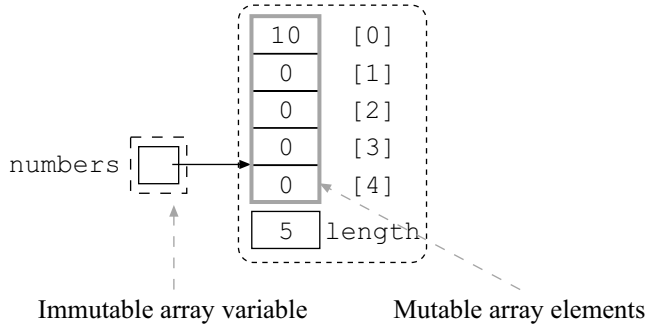


Figure 7.16 An array object is referred by an immutable array variable

Besides class variables, the keyword `final` is applicable to instance variables, method parameters and local variables. Once an immutable variable is assigned a value, it is impossible to change its value. An immutable instance variable is usually declared with initialization or is initialized during the constructor execution. The effect of making a local variable immutable is that once it is assigned a value in the method, its value cannot be changed thereafter in the method. A method parameter is assigned a value copied from the method call parameter value list. If it is marked `final`, it is impossible to change its value in the method. Its value is therefore kept unchanged throughout the execution of the method.

For example, the following method definition will cause a compile-time error:

```
public void changeParam(final int param) {
   System.out.println("The value of param = " + param);
   param = 1;
   System.out.println("The value of param is set to " + param);
}
```

In the above method definition, the statement,

```
   param = 1;
```

would cause a compile-time error because the compiler software determines that the parameter `param` is marked `final` and it is therefore an error to assign a value to it in the method.

Besides `final` variables, it is possible to mark a method or a class `final`. Please refer to Appendix A for a discussion on `final` methods and `final` classes. Throughout this unit, additional but slightly less important material has been put in the appendices for your reference.

Extending classes for reusability — inheritance

Suppose that you need a payroll calculation system to calculate the monthly salary for each staff member in the company. By analysing the problem, you determine the following requirements:

1 The details of monthly salary for a staff member include name, staff number and salary.

2 The basic salary determines the salaries of most staff. The salaries of managerial grade staff are the sums of basic salaries and commissions.

Based on the above requirements, you could design two classes, `Staff1` and `Manager1` as shown in Figure 7.17.

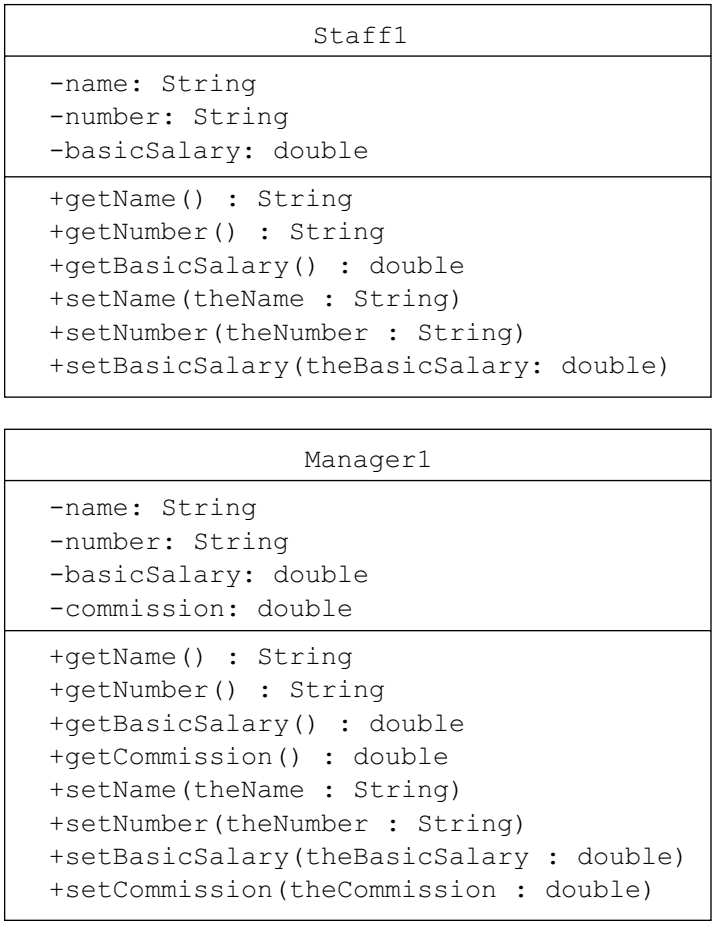


Figure 7.17 The classes Staff1 and Manager1

Based on the class designs shown in Figure 7.17, definitions of class `Staff1` and `Manager1` can be written in the Java programming language. As the class definitions are quite straightforward, they are not shown here. See page 6 on how to write setter and getter methods. Please refer to the CD-ROM or the course website for the definitions.

Comparing the designs of classes `Staff1` and `Manager1`, it is clear that large portions of the two class definitions are the same. We mentioned that it is always undesirable to have duplicated code in class definitions — if you need to modify the common code, each duplicated code needs modifying, and such operations are always error-prone and tedious.

The Java programming language enables you to reuse an existing class definition with the keyword `extends`. Figures 7.18 and 7.19 illustrate this technique using classes `Staff2` and `Manager2` respectively.

```
// Definition of class Staff2
public class Staff2 {
   // Attributes
   private String name;       //The staff name
   private String number;     //The staff number
   private double basicSalary;//The basic salary

   // Methods

   // Get the staff name
   public String getName() {
        return name;
   }

   // Get the staff number
   public String getNumber() {
        return number;
   }

   // Get the basic salary
   public double getBasicSalary() {
        return basicSalary;
   }

   // Set the name of the staff
   public void setName(String theName) {
        name = theName;
   }

   // Set the number of the staff
   public void setNumber(String theNumber) {
        number = theNumber;
   }

   // Set the salary of the staff
   public void setBasicSalary(double theBasicSalary){
        basicSalary = theBasicSalary;
   }
}
```

Figure 7.18 Staff2.java

```
// Definition of class Manager2
public class Manager2 extends Staff2 {
    // Attributes
    private double commission;      // The commission

    // Methods

    // Get the commission
    public double getCommission() {
        return commission;
    }

    // Set the commission
    public void setCommission(double theCommission) {
        commission = theCommission;
    }
}
```

Figure 7.19 Manager2.java

The definitions of the classes `Staff1` and `Staff2` are the same except for the class name. Even though the definition of the class `Manager2` is much shorter than that of the class `Manager1`, the classes `Manager1` and `Manager2` are practically equivalent. Then, how about the declaration of the attributes of `name`, `number` and `basicSalary` for the class `Manager2` and their getter/setter methods?

The definition of the class `Manager2`:

```
public class Manager2 extends Staff2 {
    ......
}
```

includes an `extends` clause, which indicates the definition of the class `Manager2` extends that of the class `Staff2`. As a result, the definition of class `Manager2` has all attributes and methods that `Staff2` defines. It is therefore unnecessary for the `Manager2` class definition to redefine those attributes and methods. Instead, the `Manager2` class definition just needs to define the extra attributes and methods that are specific to the class itself. Hence, the definition of the `Manager2` class defines the attribute `commission` and the corresponding getter/setter methods only.

To illustrate such a scenario, the designs of the class `Staff2` and `Manager2` are usually drawn in a way to highlight the extension as shown in Figure 7.20.

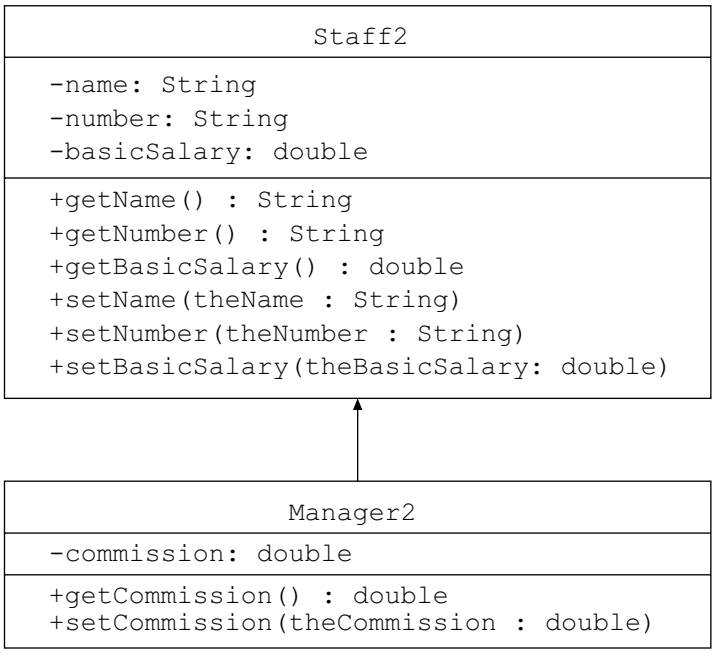


Figure 7.20 The designs for classes `Staff2` and `Manager2`

The arrow in Figure 7.20 pointing from `Manager2` class to `Staff2` class indicates that `Manager2` is an extension of the `Staff2` class.

Now, it is possible to use the `Manager2` class to create objects of the class as usual, such as:

```
Manager2 manager = new Manager2();
```

A `Manager2` object can accept message types corresponding to the behaviours defined by the `Manager2` class. For example, the following statements are valid with respect to the class definition shown in Figure 7.20 above.

```
manager.setCommission(30000.0);
double commission = manager.getCommission();
```

Also, as it is an extension of the `Staff2` class, it has all attributes and methods defined in the `Staff2` class and can therefore accept all message types that a `Staff2` object can accept. Therefore, the following statements are all valid as well:

```
manager.setName("Peter CHAN");
manager.setNumber("EDP0001");
manager.setBasicSalary(25000.0);
System.out.println("Name : " + manager.getName());
System.out.println("Number : " + manager.getNumber());
System.out.println("Basic Salary : " + manager.getBasicSalary());
```

Implementing a class by extending an existing class not only minimizes duplicated code but also provides a better way to maintain the source codes. For example, if it is necessary to specify the payment method for each staff member, an instance variable, say `paymentMethod`, and the corresponding getter/setter methods have to be added to both definitions

of classes `Staff1` and `Manager1`. However, to make such changes to the definitions of `Staff2` and `Manager2` as shown in Figure 7.18 and 7.19, it is only necessary to modify the `Staff2` class definition. As the definition of the `Manager2` class is an extension of the `Staff2` class, the changes to the definition of `Staff2` class will propagate to that of the `Manager2` class. Such a feature greatly minimizes the need to manually synchronize the duplicated code in various class definitions.

In general, if a class `ClassB` extends another class `ClassA`, it has all attributes and methods that `ClassA` class defines. In other words, we can consider that the class `ClassB` inherits all attributes and methods from the class `ClassA`. Such a feature is known as *inheritance* in the object-oriented paradigm. We call the class `ClassA` the *superclass*, parent class or base class of the class `ClassB`; and the class `ClassB` is considered to be the *subclass*, child class or derived class of the class `ClassA`.

The `extends` clause of a class definition is optional. If the `extends` clause is missing, the class implicitly extends the class `java.lang.Object` (or simply `Object` class). Therefore, the following two class definitions are equivalent:

```
public class TicketCounter {
    ......
}
```

```
public class TicketCounter extends Object {
    ......
}
```

As a result, the class relationship shown in Figure 7.20 is actually as shown in Figure 7.21.

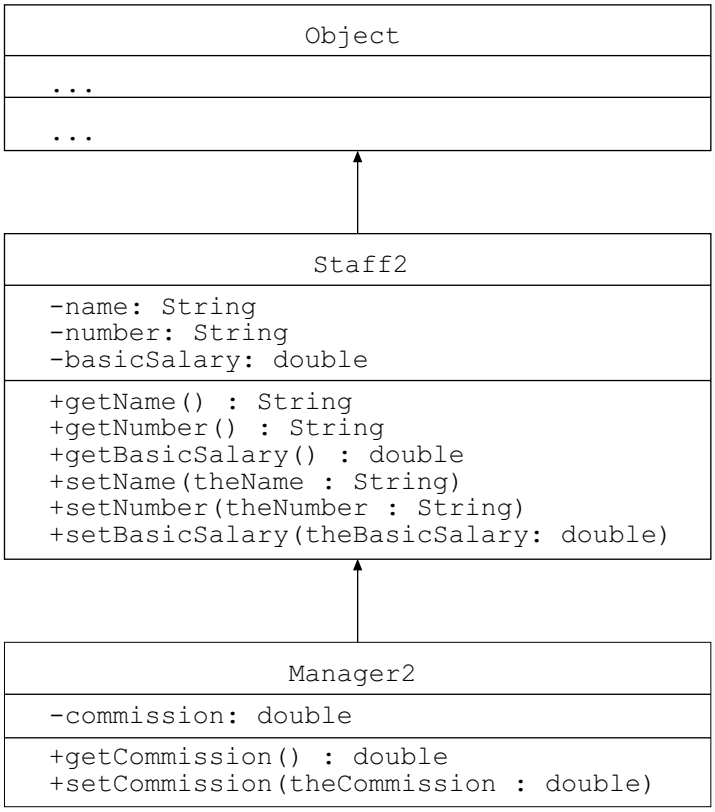


Figure 7.21 The inheritance relationship among `Object`, `Staff2` and `Manager2`

Therefore, the `Object` class is directly or indirectly the parent class of all classes in the Java programming language.

The 'is-a' relationship due to inheritance

The inheritance relationship between the classes `Staff2` and `Manager2` not only enables the definition of the `Manager2` class to extend that of the `Staff2` class but also reveals a conceptual relationship that `Manager2` '*is a*' specific type of the general type `Staff2`. In other words, a `Manager2` object can be treated as a specific type of `Staff2` object. The inheritance relationship implies an 'is a' relationship between classes `Staff2` and `Manager2`. The relationship testifies a trivial idea, 'a manager is a staff member'. However, the reverse is not true, because a staff member is not necessarily a manager.

In *Unit 2*, we mentioned that many real-world objects exhibit relationships among themselves. Some objects can be treated as a specific type of another type. For example, a bird is a specific type of animal. In line with the above discussion, if there is an existing definition of class `Animal`, it is a natural consequence that we can use the inheritance relationship between the classes `Animal` and `Bird` to write the definition of class `Bird` as the extension of class `Animal`. Then, the definition of class `Bird` can be greatly simplified by only defining its specific attributes and behaviours. This is shown in Figure 7.22.

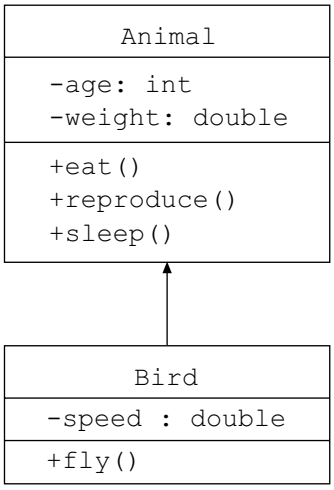


Figure 7.22 The designs of classes `Animal` and `Bird`

All animals have attributes like `age` and `weight`, and behaviours like `eat`, `reproduce` and `sleep`. A bird is a specific type of animal and has all attributes and behaviours an animal has. Also, it has its own attributes and behaviours. In Figure 7.22, it is shown that an attribute `speed` and a behaviour `fly` are specific to birds.

The above illustrates the reason why we discussed the relationships between specific types and general types in *Unit 2*. While you are analysing the requirements of a software system and a list of candidate objects (or classes) are identified, you should determine whether some classes are the specific type of another class. Such findings can help you determine whether there are inheritance relationships among them, and hence you can use the relationships for better implementations.

Additional introductory material about inheritance can be found in the following reading.

Reading

King, section 11.1, pp. 444–46

The ‘has-a’ relationship between classes

As discussed in *Unit 2*, a list of candidate objects (or classes) is identified in the analysis phase of the software development cycle. Besides analysing whether a class is the specific type of another class by exploring whether there are ‘is a’ relationships (and hence inheritance), we can determine whether there are ‘has a’ relationships.

The ‘has a’ relationship concerns the possession of an object by another one. For example, we discussed the problem of modelling a washing machine in *Unit 2*. A washing machine has an incoming valve, a drum motor and an outgoing valve. We can then determine that the individual

relationships between the class `WashingMachine` and the classes `IncomingValve`, `DrumMotor` and `OutgoingValve` are 'has a' relationships. That is:

1 A `WashingMachine` object has an `IncomingValve` object.

2 A `WashingMachine` object has a `DrumMotor` object.

3 A `WashingMachine` object has an `OutgoingValve` object.

The significance of the 'has a' relationship to a class design is that if a `ClassA` object has a `ClassB` object, the definition of the class `ClassA` defines an instance variable of type `ClassB` to refer to a `ClassB` object. With respect to the problem of a washing machine, the `WashingMachine` class is designed as shown in Figure 7.23.

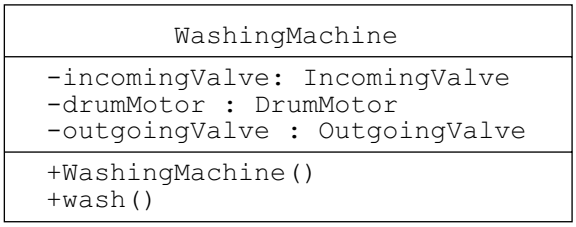


Figure 7.23 The design of class WashingMachine

We can further investigate whether the objects being possessed are an indispensable part of the class that contains them. This factor discloses the way to set up the relationship when the software is executed. For example, a drum motor is an indispensable and core part of a washing machine, because a washing machine without a drum motor does not work properly.

For such tight correlation among the classes `WashingMachine` and `DrumMotor`, a `DrumMotor` object should be created and accessible by the `WashingMachine` object when the `WashingMachine` object is created. It is true for the classes `IncomingValve` and `OutgoingValve` as well.

Therefore, a possible implementation can be written as the `WashingMachine1` class as shown in Figure 7.24.

```
// Definition of class WashingMachine1
public class WashingMachine1 {
    // Attributes
    private IncomingValve incomingValve; // the incoming valve
    private OutgoingValve outgoingValve; // the outgoing valve
    private DrumMotor drumMotor;         // the drum motor

    // Constructor
    public WashingMachine1() {
        // Create the components
        incomingValve = new IncomingValve();
        outgoingValve = new OutgoingValve();
        drumMotor = new DrumMotor();
```

```
    }

    // Methods

    // Wash the clothings
    public void wash() {
        // Washing
        incomingValve.open(5);
        drumMotor.startNormalSpeed(15);
        outgoingValve.open();
        drumMotor.startHighSpeed(5);
        outgoingValve.close();
        // Washing again with fresh water
        incomingValve.open(5);
        drumMotor.startNormalSpeed(15);
        outgoingValve.open();
        drumMotor.startHighSpeed(5);
        outgoingValve.close();
    }
}
```

Figure 7.24 WashingMachine1.java

When an object of the `WashingMachine` class is created, the scenario of the objects created as shown in Figure 7.25 is set up.

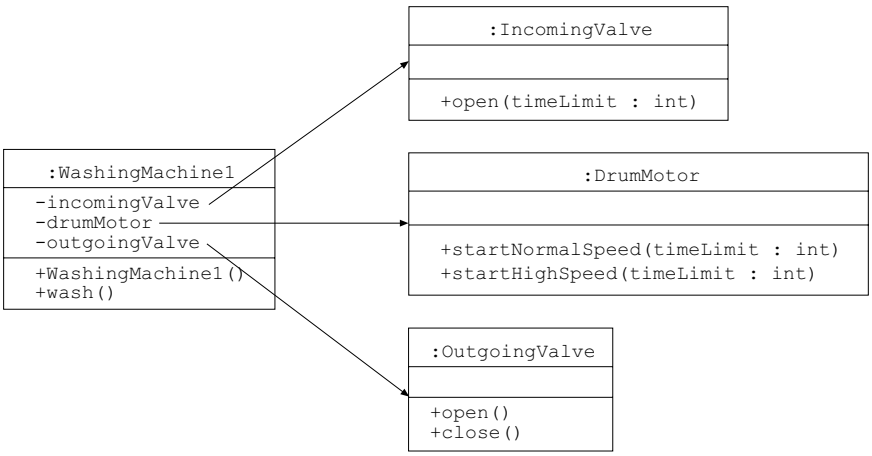


Figure 7.25 The `WashingMachine1` object with its component objects

The constructor of the `WashingMachine1` class creates objects of the classes `IncomingValve`, `DrumMotor` and `OutgoingValve` respectively. These three objects can be considered the component objects of the container `WashingMachine1` object, and the `WashingMachine1` object has attributes to refer to them.

You should consider whether the contained or possessed objects could be accessed by other objects and whether they can be updated. If the contained objects can be accessed by other objects, the container object should provide the necessary behaviours and the class should define the corresponding methods. The `WashingMachine1` class can be further enhanced to be the `WashingMachine2` class shown in Figure 7.26:

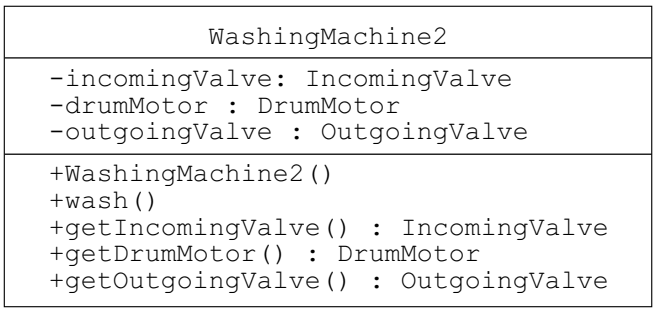


Figure 7.26 The WashingMachine2 class design

The `WashingMachine2` class defines getter methods for the attributes but no setter methods. The implication is that the attribute values cannot be changed. That is, the component objects possessed by the `WashingMachine2` object cannot be changed. The object of such a class can be considered to be immutable. If a class provides methods that can update its attribute values, the object of the class is known as a mutable object.

If the component objects are optional or can be changed during the lifespan of the container class object, the corresponding attribute of the component objects may not be initialized when the container object is created, and a corresponding setter method should be provided for updating the component objects. For example, the mouse of a computer is optional and can be changed any time, even when the computer is on. Therefore, the design of the `Computer` class can be designed as shown in Figure 7.27.

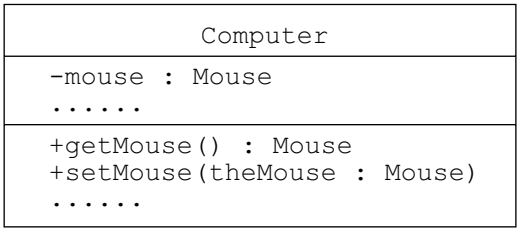


Figure 7.27 The design of class Computer

The implementation of the `Computer` class can be written as:

```
public class Computer {
    // Attributes
    private Mouse mouse;
    ......

    // Behaviours
    public Mouse getMouse() {
        return mouse;
    }
    public void setMouse(Mouse theMouse) {
        mouse = theMouse;
    }

    ......
}
```

At the time a `Computer` object is created, the attribute `mouse` is not initialized, and its values can be retrieved and updated any time. As the attribute value might be neither initialized nor set properly, it is necessary to check whether the attribute value, the attribute `mouse` in this instance, is properly set whenever its value is used, such as:

```
if (mouse != null) {
    posX = mouse.getX();
    posY = mouse.getY();
}
```

The two statements in the `if` part of the `if` statement are executed only if the attribute `mouse` is set properly, to safeguard potential runtime errors.

The significance of a mouse to a computer is not as important as that of a drum motor to a washing machine. Therefore, the ways of setting up the container and component objects are different. When you design the class, you should take this factor into account.

In the section ‘Example program — calculating the total price of items at the cashier’s counter’ in *Unit 5*, you can figure out a ‘has a’ relationship among the involved classes. A `PurchaseOrder` object has a list of `Item` objects, which is implemented as an array of `Item` object. As a result, the `PurchaseOrder` class was implemented as:

```
// Definition of class PurchaseOrder
public class PurchaseOrder {
    // Attributes
    private Item[] items;
    private int itemCount;

    // Behaviours
    // The constructor
    public PurchaseOrder(int itemTotal) {
          items = new Item[itemTotal];
   }
    ......
}
```

The array object is created while executing the `PurchaseOrder` constructor. As the array object is not to be changed and it is not necessary to retrieve its reference, there is no getter/setter method for the attribute. Instead, `Item` objects are added to the `PurchaseOrder` object one by one. As a *de facto* naming convention, a method `addItem()` is used to add an `Item` object to the `PurchaseOrder` object, and a `PurchaseOrder` object can have more than one `Item` object. If it is necessary to remove an `Item` object from the `PurchaseOrder`, another method usually named `removeItem()` is provided.

Self-test 7.3

1 Determine whether the individual relationships between/among the following sets of objects are ‘is a’ or ‘has a’.

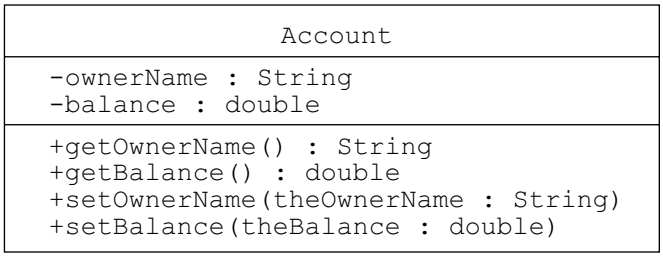
   a Car and Engine

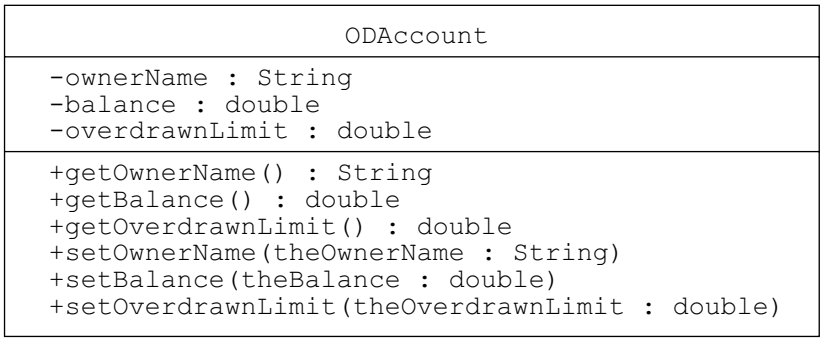
   b Computer and Personal Digital Assistant (PDA)

   c Cat, Dog and Pet

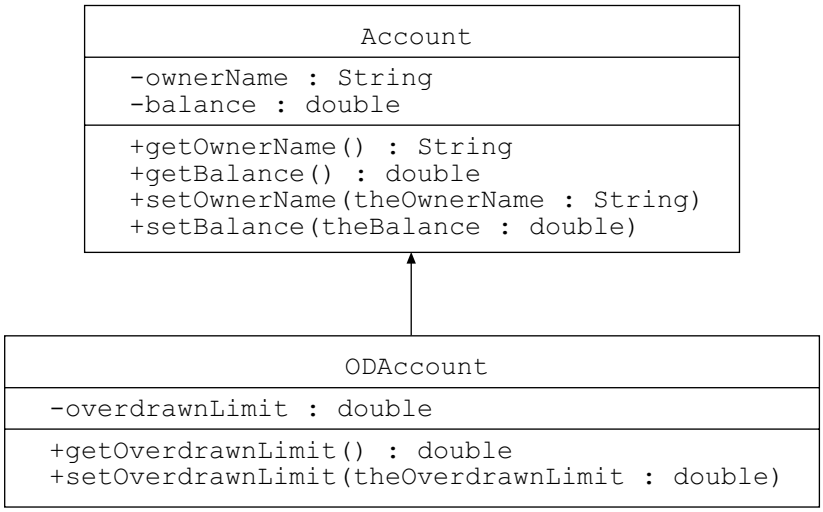
   d Shirt, Skirt and Clothing

2 In a banking software application, two classes `Account` and `ODAccount` are derived for modelling a general account and an account that can be overdrawn with a pre-defined limited. The class designs are:





The overlapping of the class designs and our understanding of the two account types suggest that the `ODAccount` class is a specific type of the general class `Account`. Therefore, it is possible to enhance the class design because of the ‘is a’ relationship as:



Based on the above class design, write the definitions of classes `Account` and `ODAccount`.

Extending classes for modifying behaviours — *inheritance* and *polymorphism*

You learned that an object performs its behaviour whenever it receives a message, and it is usually necessary to supply supplementary data to an object so that it has sufficient data to perform the operations accordingly. The corresponding implementation in the Java programming language is a method with a suitable parameter list. Occasionally, you find that an object needs to perform the same behaviour with different sets of supplementary data. This means that the object must accept a message type with different sets of supplementary data. The implication is that the corresponding class definition defines more than one method with the same name but with different parameter lists. This is possible and is known as *overloading*. We will discuss this feature very soon.

In the previous section, we discussed a feature of object-oriented programming language named inheritance. By using such a feature, we can create new classes based on an existing one and add new attributes and methods that are specific to the new classes only. Also, it is possible to derive the subclasses that have all behaviours inherited from the superclass, but the subclass objects can perform the behaviours differently. The Java programming language makes such capability possible by a feature named *overriding*, discussed later.

As subclasses of the same superclass inherit the same set of methods, objects of these subclasses and the common superclass have the methods defined in the superclass. As well as from simply using the methods in the superclass, the subclasses may redefine the methods in their own ways. Consequently, if any object of these classes accepts the same message, they may behave differently. Such a feature is known as *polymorphism*.

While writing a class definition, it is sometimes necessary to refer explicitly to members defined in the same class. Then, we can use the keyword `this`. Furthermore, if a subclass must refer to a member in the superclass, we can use another keyword `super`. We discuss the usages of the keywords `this` and `super` in detail in the following sections.

Behaviours under different situations — overloading methods

An object performs a behaviour if it receives the corresponding message. When the object performs the behaviour, it usually requires some supplementary data. Sometimes, the object is expected to perform the same behaviour under different situations. That is, the number and/or the types of supplementary data are different. From the implementation point of view, since a method name determines the object behaviour and its method parameter list specifies the supplementary data to be supplied to

the object, it is expected that a class definition permits more than one method named the same but with different parameter lists. This is known as *overloading*.

We have actually used methods that feature overloading. The one we have used the most is the `println()` method. This method is defined in the class `java.io.PrintStream`, which is provided by the software library that comes with the J2SDK. For example, to display the contents of a variable named `var`, it is possible to use the statement,

```
System.out.println(var);
```

disregarding the type of the variable `var`. The variable can be of primitive or non-primitive type (including array type). It is understood that the `println()` method displays the contents of a variable. The expected operation is the same, but the type of the supplementary data can be different.

According to the definition of the class `java.io.PrintStream`, there are ten `println()` methods in total but with different parameter lists. They are:

```
public void println() { ... }
public void println(boolean x) { ... }
public void println(char x) { ... }
public void println(int x) { ... }
public void println(long x) { ... }
public void println(float x) { ... }
public void println(double x) { ... }
public void println(char[] x) { ... }
public void println(java.lang.String x) { ... }
public void println(java.lang.Object x) { ... }
```

The ten `println()` methods have different parameter lists. One of them accepts no parameter, which shows nothing but ensures the next message to be shown is on a new line on the screen. Six methods accept a primitive-type parameter. The others accept a parameter of three commonly used non-primitive types `char[]`, `String` and `Object`. If the Java programming language did not support overloading, the above methods would have been:

```
public void println() { ... }
public void printBooleanln(boolean x) { ... }
public void printCharln(char x) { ... }
public void printIntln(int x) { ... }
public void printLongln(long x) { ... }
public void printFloatln(float x) { ... }
public void printDoubleln(double x) { ... }
public void printCharArrayln(char[] x) { ... }
public void printStringln(java.lang.String x) { ... }
public void printObjectln(java.lang.Object x) { ... }
```

The Java programming language features overloading that enables a class definition to define more than one method with the same name but with different parameter lists according to the number of parameters and/or

the parameter types. The return type of the overloaded methods can be different, even though they are usually the same. With respect to the aforementioned `println()` methods, you can verify that their parameter lists are different in either the number of parameters (no parameter or one parameter) or the parameter types.

When a class definition is compiled and a statement calls a method of an object (or a class if it is a class method), the compiler software verifies whether the corresponding class defines a method of that name and its parameter list is compatible. For example, the following statement,

```
System.out.println(100);
```

displays the value `100` on the screen. A literal `100` is by default an `int` value. Therefore, the compiler verifies whether there is a `println()` method that can accept a parameter of type `int`. As the method definition,

```
public void println(int x) { ... }
```

is found, the compilation succeeds and the above `println()` method is the one to be executed at runtime. For the following statement,

```
double d=3.14;
System.out.println(d);
```

the compiler software can figure out the type of the value to be supplied to the `println()` method is `double`. Therefore, the `println()` method that accepts a parameter of type `double` will be executed at runtime.

From a logical point of view, an object can accept the same message type with different supplementary data. However, from an implementation perspective, the corresponding methods with different parameter lists are actually different methods. They are treated as usual methods, and there is no relationship among them.

Under some situations, the types of the parameter value supplied to a method do not have to exactly match the parameter types specified in the method definition. If the compiler software determines there is a mismatch but it is possible to convert the parameter values to match the parameter types, it will try to accept the method call by implicit casting. For further details and examples, please refer to Appendix C.

Overloading methods

The overloading feature makes class design and implementation more flexible. For example, we need a method to find the maximum value between two integers. A method `findMax()` can be written as:

```
    public static int findMax(int num1, int num2) {
        int result = num1;
        if (num2 > num1) {
            result = num2;
```

```
        }
        return result;
    }
```

The method accepts two parameters of type `int` and returns the value of the larger one. As the method is utility-like and does not involve any object attribute, it is preferable to mark it `static` as a class method, so that it is not necessary to create an object before calling the method. Please refer to *Unit 3* for further details.

If it is further required to have a method to find the maximum value among three integers, we can define another `findMax()` method that accepts three parameters of type `int`, such as:

```
public static int findMax(int num1, int num2, int num3){
  int result = num1;
  if (num2 > num1 && num2 > num3) {
      result = num2;
  }
  else if (num3 > num1 && num3 > num2) {
      result = num3;
  }
  return result;
}
```

Besides the above implementation, it is possible to use the `findMax()` method that accepts two parameters of type `int`, such as:

```
public static int findMax(int num1, int num2, int num3){
   return findMax(findMax(num1, num2), num3);
}
```

The compiler software can determine that the method calls to the method `findMax` in `findMax()` method with three parameters of type `int` are actually the `findMax()` method with two parameters of type `int`.

Similarly, it is possible to define a `findMax()` method that can accept four parameters of type `int` that uses the `findMax()` methods that accept two or three parameters of type `int`, such as

```
public static int findMax(int num1, int num2, int num3, int num4){
   return findMax(findMax(num1, num2), findMax(num3, num4));
}
```

or

```
public static int findMax(int num1, int num2, int num3, int num4){
   return findMax(findMax(num1, num2, num3), num4);
}
```

The above `findMax()` methods are defined in the class `MaxFinder` as shown in Figure 7.28.

```
// Definition of class MaxFinder
public class MaxFinder {
    // Determine the maximum value of two integers
    public static int findMax(int num1, int num2) {
        int result = num1;
        if (num2 > num1) {
            result = num2;
        }
        return result;
    }

    // Determine the maximum value of three integers
    public static int findMax(int num1, int num2, int num3) {
        return findMax(findMax(num1, num2), num3);
    }

    // Determine the maximum value of four integers
    public static int findMax(int num1, int num2, int num3, int num4){
        return findMax(findMax(num1, num2, num3), num4);
    }
}
```

Figure 7.28 MaxFinder.java

To test the class `MaxFinder`, another class `TestMaxFinder` is written as shown in Figure 7.29.

```
// Definition of class TestMaxFinder
public class TestMaxFinder {

  // Main executive method
  public static void main(String args[]) {
    // Determine the maximum values and display them on the screen
    System.out.println(MaxFinder.findMax(1, 2));
    System.out.println(MaxFinder.findMax(2, 1, 3));
    System.out.println(MaxFinder.findMax(4, 3, 1, 2));
  }
}
```

Figure 7.29 TestMaxFinder.java

In the `main()` method of the `TestMaxFinder` class, different numbers of parameters of type `int` are supplied to the `findMax()` methods, to determine the maximum values of the supplied values. The return values are shown on the screen.

Each character is assigned an integral value or code, and it is therefore possible to compare two characters. To determine a character with a higher code of two, we can define another overloaded `findMax()` method that accepts two parameter values of type `char` as parameters. Furthermore, the return type of the method is `char` as a natural consequence.

```
public static char findMax(char c1, char c2) {
    char result = c1;
    if (c2 > c1) {
        result = c2;
    }
    return result;
}
```

Similarly, it is possible to write other overloaded `findMax()` methods. Each accepts two data items of a primitive type other than the type `boolean`. There are seven possible `findMax()` methods. They are defined in the class `MaxOf2Finder` as shown in Figure 7.30.

```
// Definition of class MaxOf2Finder
public class MaxOf2Finder {
    // Determine the maximum value of two bytes
    public static byte findMax(byte val1, byte val2) {
        byte result = val1;
        if (val2> val1) {
            result = val2;
        }
        return result;
    }

    // Determine the maximum value of two shorts
    public static short findMax(short val1, short val2) {
        short result = val1;
        if (val2> val1) {
            result = val2;
        }
        return result;
    }

    // Determine the maximum value of two char
    public static char findMax(char val1, char val2) {
        char result = val1;
        if (val2> val1) {
            result = val2;
        }
        return result;
    }

    // Determine the maximum value of two ints
    public static int findMax(int val1, int val2) {
        int result = val1;
        if (val2> val1) {
            result = val2;
        }
        return result;
    }

    // Determine the maximum value of two longs
    public static long findMax(long val1, long val2) {
        long result = val1;
        if (val2> val1) {
            result = val2;
```

```
        }
        return result;
    }

    // Determine the maximum value of two floats
    public static float findMax(float val1, float val2) {
        float result = val1;
        if (val2> val1) {
            result = val2;
        }
        return result;
    }

    // Determine the maximum value of two doubles
    public static double findMax(double val1, double val2) {
        double result = val1;
        if (val2> val1) {
            result = val2;
        }
        return result;
    }
}
```

Figure 7.30 MaxOf2Finder.java

Another class `TestMaxOf2Finder` is written to test the class `MaxOf2Finder.` The definition is shown in Figure 7.31.

```
// Definition of class TestMaxOf2Finder
public class TestMaxOf2Finder {

    // Main executive method
    public static void main(String args[]) {
        // Determine the maximum values and display them on the screen
        System.out.println(MaxOf2Finder.findMax((byte) 65, (byte) 66));
        System.out.println(MaxOf2Finder.findMax((short) 12,(short) 23));
        System.out.println(MaxOf2Finder.findMax('a', 'z'));
        System.out.println(MaxOf2Finder.findMax(1, 2));
        System.out.println(MaxOf2Finder.findMax(0L, -1L));
        System.out.println(MaxOf2Finder.findMax(1.5F, 2.0F));
        System.out.println(MaxOf2Finder.findMax(3.1415, 1.414));
    }
}
```

Figure 7.31 TestMaxOf2Finder.java

Compile the classes and execute the `TestMaxOf2Finder` program.
The following output is shown on the screen:

```
66
23
z
2
0
```

```
2.0
3.1415
```

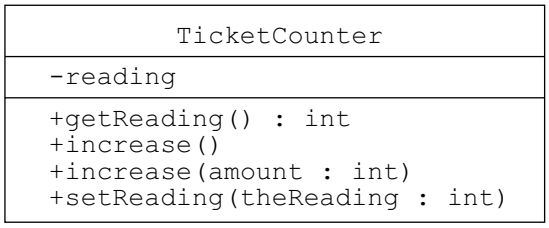
The seven `findMax()` methods of the class `MaxOf2Finder` can be substituted by the following expression:

```
(val1 > val2) ? val1 : val2
```

If the types of the variables `val1` and `val2` are both `byte`, the type of the entire expression is `byte`. Then again, if the types of the variables `val1` and `val2` are both `double`, the type of the entire expression is `double`. You can see that the `>` operator and `?:` operator can accept different data types and return a value of the suitable type. This shows that some operators provided by the Java programming language support overloading implicitly.

Self-test 7.4

Modify the definition of class `TicketCounter` we discussed in *Unit 3* so that it defines two `increase()` methods. One of them accepts no parameter and increases the attribute `reading` by one. Another one accepts a parameter `amount` to be added to the attribute `reading`. That is, the design of the class `TicketCounter` is:



Modify the `TestCounter` class presented in *Unit 3* to test the overloaded `increase()` method of the modified `TicketCounter` class.

Overloading constructors

You learned that a constructor is the special method to be executed during the creation process of an object. Like the usual methods, it is possible to write overloaded constructors. That is, a class can define more than one constructor and there is more than one way to create an object of the class.

The rules governing the method overloading are applicable to the overloading of constructors. It is possible to have more than one constructor, provided that the parameter lists are different in number of parameters and/or parameter types. For usual methods, the return value

can be different. However, this is not a concern for constructors, because a constructor returns no value.

We discussed the definition of class `TicketCounter2` as shown in Figure 7.15. It has a single constructor that accepts a parameter `theReading` of type `int`. We also mentioned that if a class defines its own constructor, the compiler software would not implicitly add a default constructor to it, which means that the following statement is illegal:

```
new TicketCounter2();
```

Empty parentheses specify the `TicketCounter2` constructor with empty parameter list.

The reason is that the `TicketCounter2` class defines no constructor with an empty parameter list. With overloading, we can provide another constructor with an empty parameter list so that the above statement becomes possible. The definition of the `TicketCounter3` class is derived based on the `TicketCounter2` class and is shown in Figure 7.32.

```
// Definition of class TicketCounter3
public class TicketCounter3 {
   // Attribute
   public static final int LIMIT = 1000;
   private int reading;    // The reading of the counter

   // The constructors
   public TicketCounter3() {
       reading = 0;
   }

   public TicketCounter3(int theReading) {
      reading = theReading % LIMIT;
   }

   // Increase the reading by one
   public void increase() {
      reading = (++reading) % LIMIT;
   }

   // Increase the reading by the specified amount
   public void increaseByAmount(int amount) {
      reading = (reading + amount) % LIMIT;
   }

   // Retrieve the value of the attribute reading
   public int getReading() {
      return reading;
   }
}
```

Figure 7.32 TicketCounter3.java

The `TicketCounter3` class defines two constructors, and their parameter lists are different. One of them has empty parameter list, and another one accepts a parameter of type `int`. Therefore, both statements in the following program segment are valid:

```
TicketCounter3 counter1 = new TicketCounter3();
TicketCounter3 counter2 = new TicketCounter3(100);
```

The first statement creates a `TicketCounter3` object with an empty parameter list. The constructor of the `TicketCounter3` class with the empty parameter list is executed during the creation process, and the attribute `reading` is initialized to be zero.

For the second statement, the parameter value list has a value `100,` which is by default a value of type `int.` When the `TicketCounter3` object is created, the constructor to be called is the one with a parameter of type `int`. The statement in the constructor assigns the value `100` (the value of the constructor parameter `theReading`) to the object attribute `reading`. As a result, after the `TicketCounter3` object is created, the value of the attribute `reading` is `100`.

Both constructors of the class `TicketCounter3` initialize the attribute `reading`. After executing the two statements that create the object, the scenario is visualized in Figure 7.33.

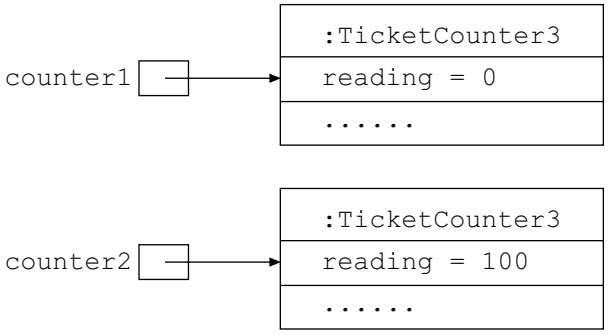


Figure 7.33 The TicketCounter3 objects after creation

Besides initializing object attributes, constructors are usually used to set up their resources and relationships with other objects. Therefore, the operations of the overloaded constructors are usually pretty much the same, except the data obtained via parameters are different.

Example: Factorial

For example, let's revisit a definition of the class `FactorialCalculator,` discussed in *Unit 4*. The `FactorialCalculator` class defines a `factorial()` method to calculate the factorial of a number supplied to it with a `do/while` loop. If the software needs to calculate the factorials of a few numbers repeatedly, it is time-consuming to re-calculate the factorials every time. An alternative approach is to store the values of the factorials in an array object so that the `factorial()` method simply returns the values

directly from an array element. Then, it is a good idea to initialize the array objects with the factorial values during the execution of the constructor. Such an implementation is used in the definition of the `FactorialCalculator1` class shown in Figure 7.34.

```
// Definition of class FactorialCalculator1
public class FactorialCalculator1 {
    // Attributes
    private int[] storage;    // The storage for the factorials
    private int limit;        // The limit of the valid number

    // Constructors
    public FactorialCalculator1() {
        // If no parameter is provided, the limit is by default 10
        limit = 10;
        // Create the array object for storage
        storage = new int[limit + 1];
        // Calculate the factorials
        storage[0] = 1;
        for (int i=1; i <= limit; i++) {
            storage[i] = i * storage[i - 1];
        }
    }

    public FactorialCalculator1(int theLimit) {
        // The limit is obtained via the parameter
        limit = theLimit;
        // Create the array object for storage
        storage = new int[limit + 1];
        // Calculate the factorials
        storage[0] = 1;
        for (int i=1; i <= limit; i++) {
            storage[i] = i * storage[i - 1];
        }
    }

    // Find the factorial of a number with the use of an array
    public int factorial(int number) {
        // Initialize the result
        int result = -1;
        // Check if the number is within valid range
        if (number <= limit) {
            // Update the result
            result = storage[number];
        }
        else {
            // If the number is invalid, show error message
            System.out.println("Out of range!");
        }
        // Return the result
        return result;
    }
}
```

Figure 7.34 FactorialCalculator1.java

The `FactorialCalculator1` class defines two overloaded constructors. It is found that most statements in the two constructors are the same, which is not a good programming practice. A possible solution to the problem is to extract the common statements as a separate method to be called by the constructors. Such an implementation is used in the definition of `FactorialCalculator2` class shown in Figure 7.35.

```
// Definition of class FactorialCalculator2
public class FactorialCalculator2 {
    // Class variable
    private static final int DEFAULT_LIMIT = 10;

    // Attributes
    private int[] storage; // The storage for the factorials
    private int limit;  // The limit of the valid number

    // Constructors
    public FactorialCalculator2() {
        // Call the initArray method to set up the array
        // with a default value
        initArray(DEFAULT_LIMIT);
    }

    public FactorialCalculator2(int theLimit) {
        // Call the initArray method to set up the array
        // with the value obtained from the parameter
        initArray(theLimit);
    }

    // Initialize the array object with the factorials
    private void initArray(int theLimit) {
        // The limit is obtained via the parameter
        limit = theLimit;
        // Create the array object for storage
        storage = new int[limit + 1];
        // Calculate the factorials
        storage[0] = 1;
        for (int i=1; i <= limit; i++) {
            storage[i] = i * storage[i - 1];
        }
    }

    // Find the factorial of a number with the use of an array
    public int factorial(int number) {
        // Initialize the result
        int result = -1;
        // Check if the number is within valid range
        if (number <= limit) {
            // Update the result
            result = storage[number];
        }
        else {
            // If the number is invalid, show error message
            System.out.println("Out of range!");
        }
```

```
        // Return the result
        return result;
    }
}
```

Figure 7.35 FactorialCalculator2.java

A new method `initArray()` is defined for the `FactorialCalculator2` class, and the two constructors call it to initialize the array object to be used. Please notice that the method is marked `private` instead of `public` because the execution of the method is part of the initialization process. It is undesirable that other objects should request the object re-initialize itself after creation. Therefore, the method is marked `private` so that only the statements in its own class definition can call it. As a rule of thumb, the methods of the class to be called by the constructor are usually marked `private`, unless other objects will also call them.

To minimize the duplicated source codes, other than declaring a separate method, the Java programming language enables a constructor to call another constructor as part of the creation process, by using the keyword `this`.

The `this` keyword — calling an overloaded constructor

If a class defines overloaded constructors, it is possible to use the keyword `this` in a constructor to call another constructor with parameter values of suitable types. The syntax to use the keyword `this` to make a call to another constructor is:

```
this(parameter-value-list);
```

The parameter value list determines the constructor to be called, and that constructor will be executed with the parameter values supplied. Based on the definition of the class `FactorialCalculator1`, a new class `FactorialCalculator3` is defined as shown in Figure 7.36.

```
// Definition of class FactorialCalculator3
public class FactorialCalculator3 {
    // Class variable
    private static final int DEFAULT_LIMIT = 10;

    // Attributes
    private int[] storage;    // The storage for the factorials
    private int limit;        // The limit of the valid number

    // Constructors
    public FactorialCalculator3() {
        // Call another constructor with parameter of type int
        // to perform the actual operations
        this(DEFAULT_LIMIT);
    }
```

```
    public FactorialCalculator3(int theLimit) {
        // The limit is obtained via the parameter
        limit = theLimit;
        // Create the array object for storage
        storage = new int[limit + 1];
        // Calculate the factorials
        storage[0] = 1;
        for (int i=1; i <= limit; i++) {
            storage[i] = i * storage[i - 1];
        }
    }

    // Find the factorial of a number with the use of an array
    public int factorial(int number) {
        // Initialize the result
        int result = -1;
        // Check if the number is within valid range
        if (number <= limit) {
            // Update the result
            result = storage[number];
        }
        else {
            // If the number is invalid, show error message
            System.out.println("Out of range!");
        }
        // Return the result
        return result;
    }
}
```

Figure 7.36 FactorialCalculator3.java

The constructor of the class `FactorialCalculator3` with an empty parameter list calls another constructor with a parameter of type `int` to perform the actual operations. The definition is simply:

```
public FactorialCalculator3() {
    // Call another constructor with parameter of type int
    // to perform the actual operations
    this(DEFAULT_LIMIT);
}
```

The value of the class variable `DEFAULT_LIMIT` is `10` and its type is `int`. With the use of the class variable, the single statement in the constructor prepares the necessary parameter value list to be supplied to another constructor with the parameter list of an `int` parameter. As a result, if a `FactorialCalculator3` object is created with the statement,

```
new FactorialCalculator3()
```

its array object stores the factorials of numbers from 0 up to 10.

Example: Item

In Self-test 7.2, question 2 asked you to provide a constructor for the class `Item`. The constructor needs three data items of type `String`, `double` and `int`. You can add three constructors so that an object of the class can be created with less data, and default values are used for the missing ones. The definition of the class `Item1` is written in Figure 7.37.

```
// Definition of class Item1
public class Item1 {
    // Default values
    private static final String DEFAULT_NAME = "Untitled";
    private static final double DEFAULT_PRICE = 0.0;
    private static final int DEFAULT_QUANTITY = 1;

    // Attributes
    private String name;
    private double price;
    private int quantity;

    // The constructors of the class
    public Item1() {
        // Call the constructor with an extra default name
        this(DEFAULT_NAME);
    }

    public Item1(String theName) {
        // Call the constructor with an extra default price
        this(theName, DEFAULT_PRICE);
    }

    public Item1(String theName, double thePrice) {
        // Call the constructor with an extra default quantity
        this(theName, thePrice, DEFAULT_QUANTITY);
    }

    public Item1(String theName, double thePrice, int theQuantity) {
        // Set the attributes properly
          name = theName;
       price = thePrice;
       quantity = theQuantity;
   }

    // Behaviours

    // Set the attribute name
    public void setName(String theName) {
        name = theName;
    }

    // Set the attribute price
    public void setPrice(double thePrice) {
        price = thePrice;
    }
```

```
    // Set the attribute quantity
    public void setQuantity(int theQuantity) {
        quantity = theQuantity;
    }

    // Get the subtotal of this item
    public double findTotal() {
        return price * quantity;
    }
}
```

Figure 7.37 Item1.java

The `Item1` class has four constructors with different parameter lists. There are therefore four possible ways to create an `Item1` object, such as:

```
Item1 item1 = new Item1();
Item1 item2 = new Item1("Coke");
Item1 item3 = new Item1("Coke", 2.5);
Item1 item4 = new Item1("Coke", 2.5, 6);
```

If no data are supplied to the `Item1` constructor, the compiler software chooses the following constructor:

```
public Item1() {
   // Call the constructor with an extra default name
   this(DEFAULT_NAME);
}
```

With the class variable `DEFAULT_NAME` of type `String`, the statement in the constructor prepares the necessary parameter value and calls another constructor with a parameter of type `String` right away.

Similarly, the constructor with a parameter of type `String`

```
public Item1(String theName) {
    // Call the constructor with an extra default price
    this(theName, DEFAULT_PRICE);
}
```

prepares a parameter list of one more datum of the type `double` using the class variable `DEFAULT_PRICE` and calls another constructor with parameters of types `String` and `double`.

```
public Item1(String theName, double thePrice) {
    // Call the constructor with an extra default quantity
    this(theName, thePrice, DEFAULT_QUANTITY);
}
```

Like the previous two constructors, the above constructor prepares parameter values of type `String`, `double` and `int` and calls the fourth constructor that performs the actual operations that assigns the parameter values to the object attributes:

```
public Item1(String theName, double thePrice, int theQuantity) {
```

```
        // Set the attributes properly
          name = theName;
      price = thePrice;
      quantity = theQuantity;
  }
```

You should notice that if you use the `this()` statement to call another constructor of the class, the statement must be the first statement in the method or compilation fails.

In this section, the overloaded constructors with fewer parameters call another overloaded constructor with one more parameter. There is an alternative approach to write the overloaded constructors so that the constructors with more parameters call another overloaded constructor with one less parameter. Please refer to Appendix D for the details of such an approach.

Self-test 7.5

Modify the `Time3` class (as shown in Figure 7.4) so that it defines two overloaded constructors. The class design is:

Time3
-hour : int -minute : int -second : int -DEFAULT_HOUR : int = 0 -DEFAULT_MINUTE : int = 0 -DEFAULT_SECOND : int = 0
+Time2() +Time2(theHour : int, theMinute : int, theSecond : int) +getHour() : int +getMinute() : int +getSecond() : int +setHour(theHour : int) +setMinute(theMinute : int) +setSecond(theSecond : int)

(The variables `DEFAULT_HOUR`, `DEFAULT_MINUTE` and `DEFAULT_SECOND` are `final` and are class variables of the class.)

Modify the definition of the `Time2` class and add two overloaded constructors to it. The constructor without a parameter list should call another constructor and supply the default values, `DEFAULT_HOUR`, `DEFAULT_MINUTE` and `DEFAULT_SECOND`.

The `this` keyword — resolving object members explicitly

Besides being used for calling another constructor, the keyword `this` is commonly used for referring to the object itself. When an object performs its behaviours — that is, executing non-class (non-`static`)

methods — it is possible to use the keyword `this` to inquire the reference of the object itself. Please use the following reading to learn about the uses of the keyword `this`.

Reading

King, section 10.4, pp. 402–5

The above reading suggests some uses of the keyword `this`. The following section provides explanations from another perspective to help you understand its uses.

You have learned that a class definition is a blueprint of a group of objects to be created. The created objects have the same set of behaviours and attributes, but each object is an individual entity and hence a different reference. You can imagine each object has a virtual attribute `this` that is initialized to be the reference of the object itself.

As the keyword `this` can be considered an object variable that refers to the object itself, it is commonly used in the following two situations:

1 explicitly referring to the object behaviours and attributes

2 calling a method with supplementary data of the reference of the object that calls the method.

We discussed the definition of class `FactorialCalculator2` earlier. The two constructors call the `initArray()` method to initialize its array objects:

```
// Constructors
public FactorialCalculator2() {
    // Call the initArray method to set up the array
    // with a default value
    initArray(DEFAULT_LIMIT);
}

public FactorialCalculator2(int theLimit) {
    // Call the initArray method to set up the array
    // with the value obtained from the parameter
    initArray(theLimit);
}

// Initialize the array object with the factorials
private void initArray(int theLimit) {
    // The limit is obtained via the parameter
    limit = theLimit;
    // Create the array object for storage
    storage = new int[limit + 1];
    // Calculate the factorials
    storage[0] = 1;
    for (int i=1; i <= limit; i++) {
```

```
            storage[i] = i * storage[i - 1];
        }
    }
```

The `initArray()` method is called without any prefix. It is understood that the method to be called is a method of the object itself. You can consider that a method call to a method without any prefix has an implicit prefix '`this.`'. Therefore, the two constructors are interpreted to be:

```
    public FactorialCalculator2() {
        // Call the initArray method to set up the array
        // with a default value
        this.initArray(DEFAULT_LIMIT);
    }

    public FactorialCalculator2(int theLimit) {
        // Call the initArray method to set up the array
        // with the value obtained from the parameter
        this.initArray(theLimit);
    }
```

The interpretation is that a message `initArray` is sent to the object itself with supplementary data of type `int`, so that the object itself will perform the behaviour `initArray`. That is, it executes the `initArray()` method.

The same argument applies to object attributes. If a statement in a method involves a variable, it is first determined whether it is a method parameter or a local variable. If the variable name can be found in the parameter list or is referring to a local variable before the statement, it is interpreted to be the method parameter or local variable. Otherwise, it is further determined whether it is an attribute (instance variable) of the object or class variable of the class. Then, a compilation error will occur if the class definition does not declare the variable.

Like methods, a variable that is determined to be an object attribute can be considered implicitly prefixed by '`this.`'. Hence, the `initArray()` method is interpreted to be:

```
    // Initialize the array object with the factorials
    private void initArray(int theLimit) {
        // The limit is obtained via the parameter
        this.limit = theLimit;
        // Create the array object for storage
        this.storage = new int[this.limit + 1];
        // Calculate the factorials
        this.storage[0] = 1;
        for (int i=1; i <= this.limit; i++) {
            this.storage[i] = i * this.storage[i - 1];
        }
    }
```

In line with our understanding of the `.` (dot) operator, `this.limit` means the attribute `limit` of the object referred by `this`. As the

keyword `this` always refers to the object itself, `this.limit` is referring to the attribute `limit` of the object itself.

Even though the attributes `limit` and `storage` and the `initArray()` method are marked `private` in the class definition, it is possible to have expressions like `this.limit` and `this.initArray()` because an object can access its own `private` members.

The prefix '`this.`' is implicit and is determined automatically by the compiler software. If you are still curious about the interpretation of the keyword `this`, please refer to Appendix E for an experiment for testing the content of the keyword `this`.

Usually, it is unnecessary to use the prefix '`this.`'. However, in some situations it is preferable to provide the prefix yourself. For example, the constructor of the `TicketCounter4` shown in Figure 7.38 needs an explicit prefix.

```
// Definition of class TicketCounter4
public class TicketCounter4 {
   // Attribute
   public static final int LIMIT = 1000;
   private int reading;  // The reading of the counter

   // The constructors
   public TicketCounter4() {
       reading = 0;
   }

   public TicketCounter4(int reading) {
       this.reading = reading % LIMIT;
   }

   // Increase the reading by one
   public void increase() {
       reading = (++reading) % LIMIT;
   }

   // Increase the reading by the specified amount
   public void increase(int amount) {
       reading = (reading + amount) % LIMIT;
   }

   // Retrieve the value of the attribute reading
   public int getReading() {
       return reading;
   }
}
```

Figure 7.38 TicketCounter4.java

It is acceptable for a method or constructor with parameters to be named the same as attributes of the class. As mentioned, if the variable is used

in a method without a prefix, it is first determined whether it is a local variable or method parameter. Therefore, with respect to the constructor of the class `TicketCounter4`, if it were defined to be

```
public TicketCounter4(int reading) {
   reading = reading % LIMIT;
}
```

the variable `reading` on both sides of the = operator would have been referring to the parameter `reading` instead of to the object attribute `reading`; the interpretation would have been performing the calculation `reading % LIMIT` and assigning the result to the parameter `reading`. As parameters and local variables are removed from the memory as soon as the method terminates, the object attribute `reading` would never be updated.

To resolve such a problem, we can make use of the prefix '`this.`' to explicitly indicate the object attribute in the `TicketCounter4` constructor. When the constructor is called by an expression, say `new TicketCounter4(100)`, the expression `this.reading` is referring to the object attribute `reading`, and the variable `reading` by itself is referring to the parameter as illustrated in Figure 7.39.

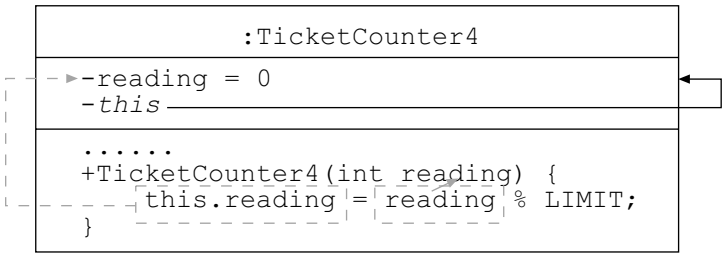


Figure 7.39 The interpretations of expressions this.reading and reading by itself

The `this` keyword — calling a method with the keyword `this` as parameter values

Another use of the keyword `this` is that the object can supply its reference to other methods for further manipulations. For example, based on the definition of the class `TicketCounter4`, we can write a `TicketCounter5` class, as shown in Figure 7.40. It provides another overloaded constructor so that we can supply a reference of an existing `TicketCounter5` object. Furthermore, the class definition provides another new method named `cloneCounter()` to be discussed very soon.

```
// Definition of class TicketCounter5
public class TicketCounter5 {
   // Attribute
   public static final int LIMIT = 1000;
   private int reading;  // The reading of the counter

   // The constructors
   public TicketCounter5() {
       reading = 0;
   }

   public TicketCounter5(int reading) {
      this.reading = reading % LIMIT;
   }

   public TicketCounter5(TicketCounter5 counter) {
       this.reading = counter.reading;
   }

   // Create a new TicketCounter5 with the same reading
   public TicketCounter5 cloneCounter() {
       return new TicketCounter5(this);
   }

   // Increase the reading by one
   public void increase() {
      reading = (++reading) % LIMIT;
   }

   // Increase the reading by the specified amount
   public void increase(int amount) {
      reading = (reading + amount) % LIMIT;
   }

   // Retrieve the value of the attribute reading
   public int getReading() {
      return reading;
   }
}
```

Figure 7.40 TicketCounter5.java

The operation of the constructor with a parameter of type `TicketCounter5` is to copy the value of the attribute `reading` of the `TicketCounter5` object referred by the parameter `counter` to the attribute `reading` of the object being created, for example, the following program segment:

```
TicketCounter5 counter1 = new TicketCounter5(100);
TicketCounter5 counter2 = new TicketCounter5(counter1);
```

The first statement in the above program segment declares a variable `counter1` of type `TicketCounter5`, creates a `TicketCounter5` object with initial reading of `100` and assigns the reference of the newly

created object to the variable `counter1`. The scenario is shown in Figure 7.41.

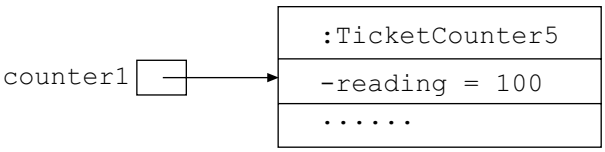


Figure 7.41 The first TicketCounter5 object

The second statement declares the variable `counter2` of the type `TicketCounter5`. During the creation process of the second `TicketCounter5` object, the constructor with the parameter of type `TicketCounter5` is chosen and called. The reference of the first `TicketCounter5` object is supplied to the constructor and is assigned to the parameter `counter`. The scenario when the constructor of the second `TicketCounter5` object is executing is shown in Figure 7.42.

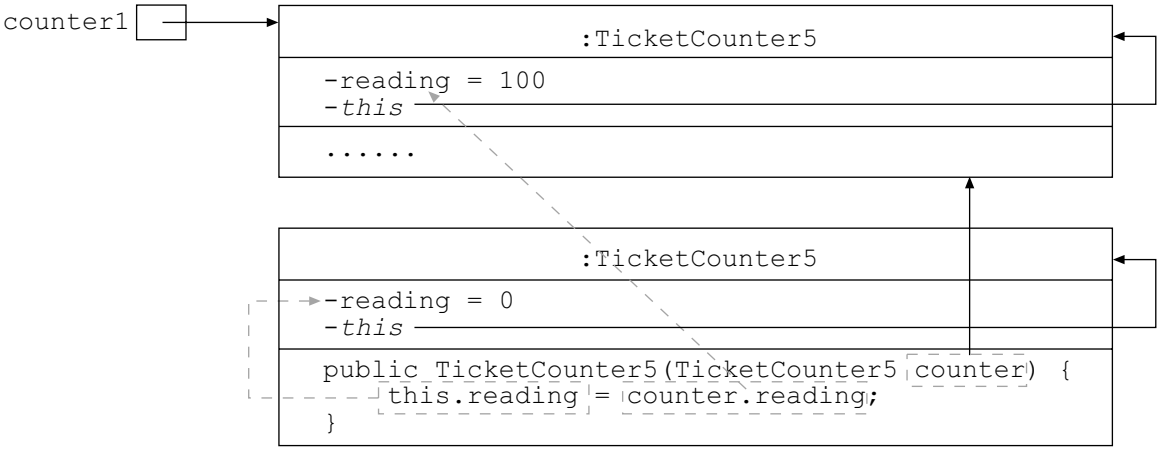


Figure 7.42 The scenario while executing the constructor of the second TicketCounter5 object

Executing the statement in the constructor

```
this.reading = counter.reading;
```

assigns the value of the attribute `reading` of the `TicketCounter5` object referred by the variable `counter` to the attribute `reading` of the new `TicketCounter5` object.

Therefore, after the execution of the constructor, the objects become as shown in Figure 7.43.

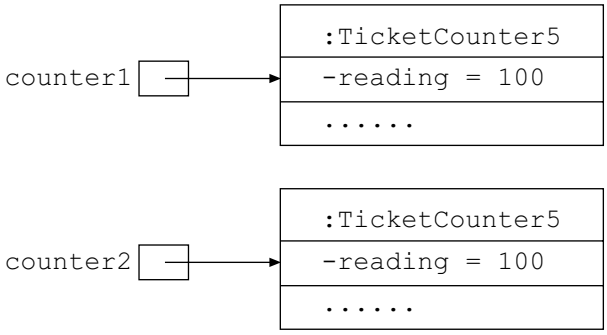


Figure 7.43 The two TicketCounter5 objects after executing the program segment

The `TicketCounter5` class provides another method that can create a new `TicketCounter5` object, and the attribute `reading` of the object is initialized to be the same as the attribute `reading` of the one that creates it. For example:

```
TicketCounter5 counter1 = new TicketCounter5(100);
TicketCounter5 counter2 = counter1.cloneCounter();
```

The above program segment might seem a little strange to you. Let's trace its operation in detail. First of all, the first statement declares a variable `counter1` of type `TicketCounter5` and refers to the created `TicketCounter5` object. For the second statement, it first declares a variable `counter2` of the type `TicketCounter5`. Then, a message `cloneCounter` is sent to the `TicketCounter5` object that is referred by the variable `counter1`. The scenario when the `cloneCounter()` method is executing can be visualized in Figure 7.44.

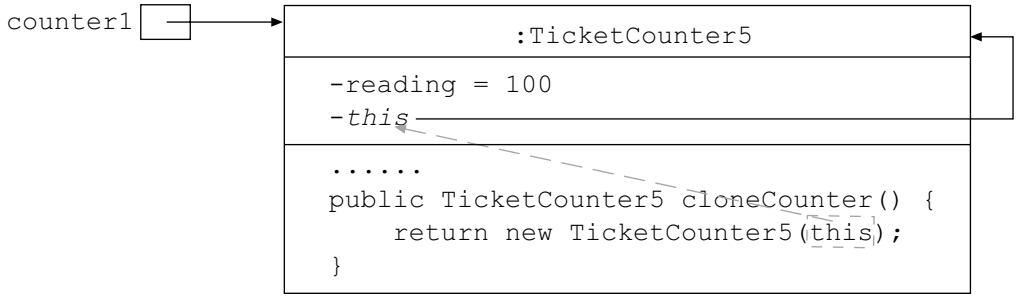


Figure 7.44 The scenario when the cloneCounter() method is executing

Executing the statement in the `cloneCounter()` method, `new TicketCounter5(this),` the reference of the `TicketCounter5` object itself, which is the first `TicketCounter5` object referred to by the `counter1` variable, and the keyword `this` in the `cloneCounter()` method is supplied to the constructor of the new `TicketCounter5` object. The type of the keyword `this` in the method is considered the type of the object that is executing the method. Therefore, in the `cloneCounter()` method, the type of the keyword `this` is treated as `TicketCounter5`. As a result, the constructor with a parameter of type `TicketCounter5` is determined and executed. The parameter `counter` is assigned the reference of the first

`TicketCounter5` object that executes the `cloneCounter()` method. The scenario is visualized in Figure 7.45.

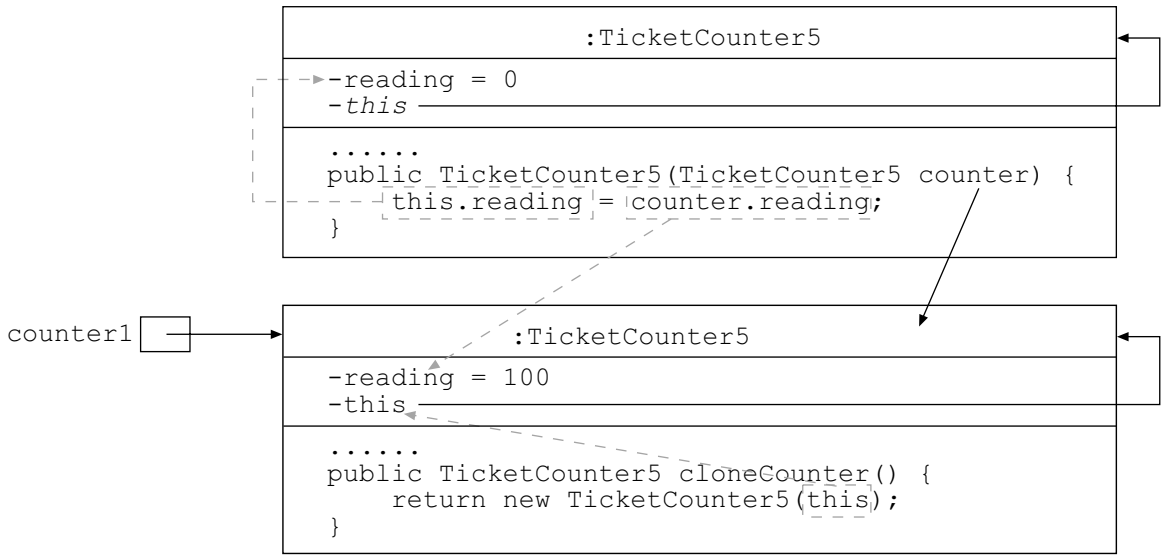


Figure 7.45 The scenario when the constructor TicketCounter5(TicketCounter5) is executed

Figure 7.45 indicates that the parameter `counter` is referring to the `TicketCounter5` object that is supplied via a parameter, which is the first `TicketCounter5` object referred by the variable `counter1`. As a result, the attribute `reading` of the new `TicketCounter5` object is initialized to be the same as that of the first `TicketCounter5` object. Therefore, executing the statement in the constructor initializes the object attribute `reading` to `100` as shown in Figure 7.46. The prefix '`this.`' is optional and is written in the `TicketCounter5` constructor to clarify the owner of the `reading` attributes.

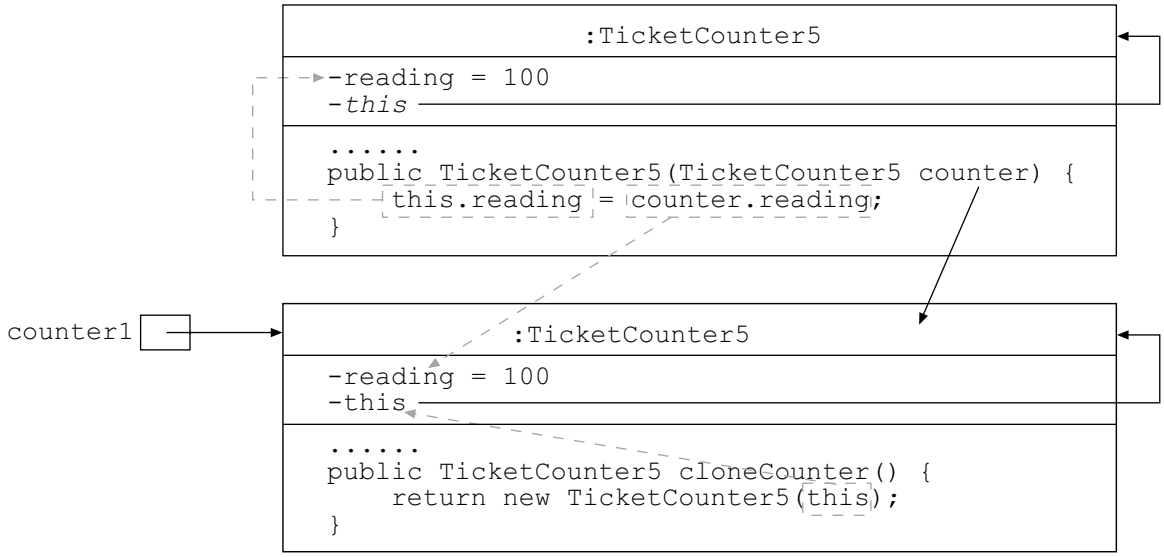


Figure 7.46 The scenario of the newly created TicketCounter5 object after executing the constructor

The flow of control and the reference to the newly created `TicketCounter5` object is returned to the `cloneCounter()` method, and the reference of the newly created `TicketCounter5` object (that matches the return type of the method) is returned right away and is assigned to the `counter2` variable. Therefore, the scenario becomes as shown in Figure 7.47.

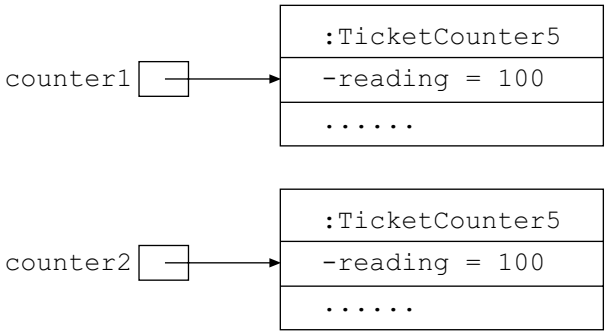


Figure 7.47 The two TicketCounter5 objects

Please use the following self-test to verify your understanding of the uses of the keyword `this`.

Self-test 7.6

1 By using the keyword `this`, modify the constructors and the setter methods of the class `Item1` so that the parameter names can be the same as the object attributes.

2 A student types the definition of the class `TicketCounter4` in an editor application, and the constructor is typed to be:

```
public TicketCounter4(int raeding) {
    this.reading = reading % LIMIT;
}
```

Will there be any compilation error? If there is no compilation error, what will happen if the above constructor is executed?

Modifying behaviours by overriding methods

In the section ‘Extending classes for reusability’, we said that it was possible to define a new class (the subclass) based on an existing class (the superclass) by adding attributes and methods. Then, the object of the subclass has the attributes and methods inherited from the superclass and those defined in the subclass itself. How about the subclass defining a method with the name and parameter list exactly the same as one of the methods defined in the superclass? Is it possible?

The answer is yes. It is possible to define a method in the subclass with a name and parameter list that are identical to a method defined in the superclass. This is known as overriding, and we discuss it in detail next. However, if a method defined in the subclass is the same as a method in the superclass but the parameter lists are *different*, it is not overriding but just overloading. Please refer to Appendix F for a discussion of this. There is an example to illustrate it.

Overriding methods in superclasses

During problem analysis, you might find that an existing class fits your requirements except for a few methods. Then, you are tempted to copy most parts of the existing class definition and write those specific methods in your own way. It is, of course, a workable method. However, the object-oriented paradigm provides a powerful facility that enables you to reuse the definition of an existing class and supersede the unsuitable parts.

For example, the definition of class `Date1` shown in Figure 7.48 can be used to represent a date.

```
// definition of the class Date1
public class Date1 {
    // Attributes
    private int day;        // the day
    private int month;      // the month
    private int year;       // the year

    // Getter methods

    // get the day
    public int getDay() {
        return day;
    }

    // get the month
    public int getMonth() {
        return month;
    }

    // get the year
    public int getYear() {
        return year;
    }

    // Setter methods

    // set the day
    public void setDay(int day) {
        this.day = day;
    }

    // set the month
    public void setMonth(int month) {
```

```
        this.month = month;
    }

    // set the year
    public void setYear(int year) {
        this.year = year;
    }

    // Show the date in yyyymmdd format on the screen
    public void print() {
        System.out.print(year);
        System.out.print((month < 10) ? "0" + month : "" + month);
        System.out.print((day < 10) ? "0" + day : "" + day);
    }
}
```

Figure 7.48 Date1.java

To test the `Date1` class, another class `TestDate1` is written in Figure 7.49.

```
// Definition of class TestDate1
public class TestDate1 {

    // Main executive method
    public static void main(String args[]) {
        Date1 date = new Date1();
        date.setYear(2003);
        date.setMonth(1);
        date.setDay(2);
        date.print();
    }
}
```

Figure 7.49 TestDate1.java

Compile the two classes and execute the `TestDate1` program. The first statement of the `main()` method of the `TestDate1` creates a `Date1` object referred by the local variable `date`. Then, the subsequent three statements initialize the attributes `year`, `month` and `day` of the `Date1` object. The final statement calls the `print()` method of the `Date1` object. According to the definition of the `print()` method of the `Date1` class, the `Date1` object shows the values of its attributes `day`, `month` and `year` on the screen:

```
20030102
```

Suppose that you are going to write another software application, and you need a class to model or represent the dates but the date has to be shown in the British style; that is, in the `dd/mm/yyyy` format in which `dd`, `mm` and `yyyy` stand for a two-digit day, two-digit month and four-digit year respectively. You'll probably find that the definition of the

`Date1` class shown in Figure 7.48 fulfils most of the requirements except the `print()` method.

The Java programming language enables you to use the keyword `extends` to extend another class definition. Besides extending the existing class definition, it is possible to override parts of it. For example, based on the existing class `Date1`, it is possible to define another class `BritishDate1` as shown in Figure 7.50.

```
// Definition of class BritishDate1
public class BritishDate1 extends Date1 {

    // Override the print() method defined in the Date1 class
    // and show the date in dd/mm/yyyy format
    public void print() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in dd/mm/yyyy format
        System.out.print((day < 10) ? "0" + day : "" + day);
        System.out.print("/");
        System.out.print((month < 10) ? "0" + month : "" + month);
        System.out.print("/");
        System.out.print(year);
    }
}
```

Figure 7.50 BritishDate1.java

Even though the class `BritishDate1` is a subclass of the class `Date1`, its `print()` method cannot access the `private` members of the class `Date1`, the `private` attributes in this case. The reason is that the access modifier `private` prevents those members from being accessed by other classes. Up to now, we have come across two access modifiers, `public` and `private`. Actually, the Java programming language provides three access modifiers to control the accessibilities of variables and methods. Although the access modifiers `public` and `private` suffice for most cases, a detailed discussion of different accessibility is provided in Appendix G for completeness.

The relationship between the classes `Date1` and `BritishDate1` is visualized in Figure 7.51.

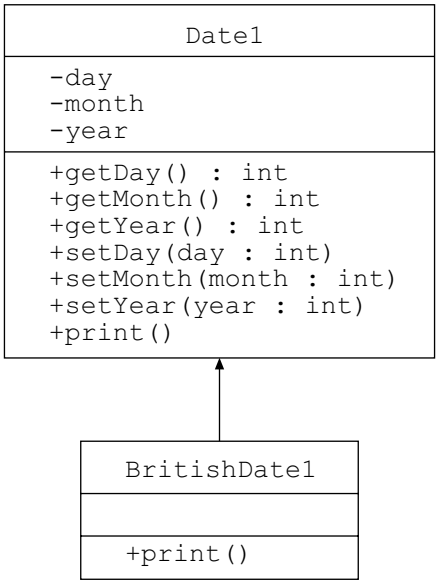


Figure 7.51 The inheritance relationship between classes Date1 and BritishDate1

A `BritishDate1` is a `Date1`. Therefore, it justifies the derivation of the `BritishDate1` class based on the `Date1` class. With the use of the `extends` clause in the class definition, the definition of the `BritishDate1` class extends that of the `Date1` class, and the `BritishDate1` inherits all methods from the superclass `Date1`. Furthermore, no constructor is defined for the `BritishDate1`, an implicit default constructor when an empty parameter list is added.

To test the `BritishDate1` class, the `TestDate1` is modified to test the `BritishDate1` instead and is written as `TestBritishDate1` in Figure 7.52.

```
// Definition of class TestBritishDate1
public class TestBritishDate1 {

    // Main executive method
    public static void main(String args[]) {
        BritishDate1 date = new BritishDate1();
        date.setYear(2003);
        date.setMonth(1);
        date.setDay(2);
        date.print();
    }
}
```

Figure 7.52 TestBritishDate1.java

Compared with the `TestDate1` class, the object to be created and manipulated is a `BritishDate1` object instead of a `Date1` object, and other statements in the `main()` method of the `TestBritishDate1` are the same as in the `main()` method of the `TestDate1` class.

Compile the classes and execute the `TestBritishDate1` program. The output to be shown on the screen is:

```
02/01/2003
```

The method calls to the methods `setYear()`, `setMonth()` and `setDay()` of the `BritishDate1` object in the `main()` method confirms that a `BritishDate1` object has these methods. Furthermore, the output shown on the screen indicates the `print()` method to be called is the one defined by the `BritishDate1` class.

Similarly, it is possible to derive another class `AmericanDate1` based on the `Date1` class so that the date to be shown on the screen is in the American style (the `m/d/yyyy` format). The definition of the class `AmericanDate1` is written as shown in Figure 7.53.

```
// Definition of class AmericanDate1
public class AmericanDate1 extends Date1 {

    // Override the print() method defined in the Date1 class
    // and show the date in m/d/yyyy format
    public void print() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in m/d/yyyy format
        System.out.print(month + "/" + day + "/" + year);
    }
}
```

Figure 7.53 AmericanDate1.java

We can write a `TestAmericanDate1` class similar to that of `TestBritishDate1` to test the `AmericanDate1` class. The definition of the class `TestAmericanDate1` is shown in Figure 7.54.

```
// Definition of class TestAmericanDate1
public class TestAmericanDate1 {
    public static void main(String args[]) {
        AmericanDate1 date = new AmericanDate1();
        date.setYear(2003);
        date.setMonth(1);
        date.setDay(2);
        date.print();
    }
}
```

Figure 7.54 TestAmericanDate1.java

Compile the classes `AmericanDate1` and `TestAmericanDate1`, and execute the `TestAmericanDate1` program. The following message is shown on the screen:

```
1/2/2003
```

The `AmericanDate1` object shows the date in the American format as expected.

You might wonder whether it is possible to derive the `AmericanDate1` class based on the `BritishDate1` class, which means that the `AmericanDate1` is implemented as a subclass of the `BritishDate1` class. The answer to this problem is twofold — the practical aspect and the conceptual aspect. It is practically feasible to do so, and the `AmericanDate1` object would work as expected, because the class `AmericanDate1` can inherit the getter/setter methods indirectly from the `Date1` class (via the `BritishDate1` class) and the `print()` method from the `BritishDate1` class. However, it is conceptually incorrect to do so, because deriving the `AmericanDate1` class based on the `BritishDate1` class implies there is inheritance or an 'is a' relationship between the two classes. However, it is illogical to say 'an `AmericanDate1` is a `BritishDate1`' but 'an `AmericanDate1` is a `Date1`'. Therefore, it is conceptually correct to derive the `AmericanDate1` class based on the `Date1` class instead of on the `BritishDate1` class.

The relationship among the classes `Date1`, `AmericanDate1` and `BritishDate1` is visualized in Figure 7.55.

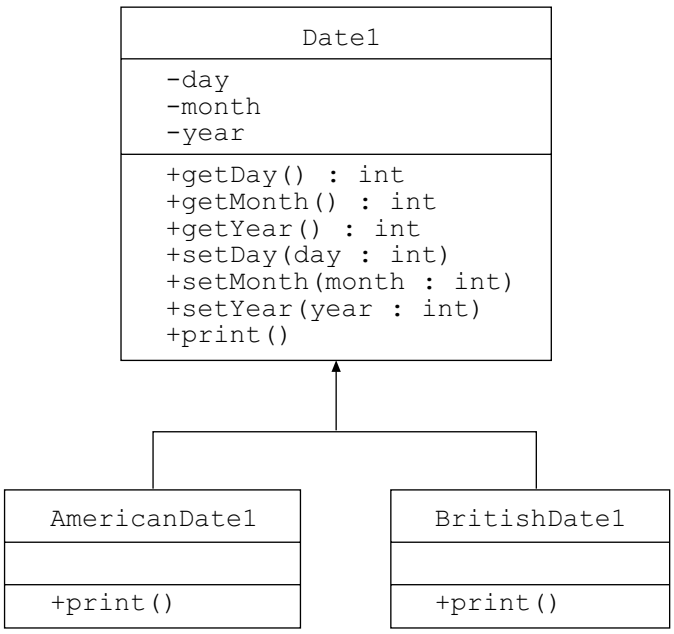


Figure 7.55 The relationships among classes Date1, AmericanDate1 and BritishDate1

You can see that inheritance can ease the way to reuse existing class definitions, and hence your programming productivity can be greatly enhanced. Furthermore, while deriving the new classes, the programmers can concentrate on the methods that need modification. The members inherited from the parent class probably have been thoroughly tested by the programmer who wrote the class. This implies that the reliability of

the software application is usually better than writing the new class from scratch.

More information about overriding can be found in the following reading.

Reading

King, section 11.3, pp. 452–53

Self-test 7.7

Define a `ChineseDate1` class as a subclass of the `Date1` class and define its own `print()` method that shows the date in the `yyyy.m.d` format, where `yyyy`, `m` and `d` are the four-digit year, one- or two-digit month and one- or two-digit day respectively.

Polymorphism

With respect to the definition of the class `TestBritishDate1` shown in Figure 7.52, the statement for creating a `BritishDate1` object is:

```
BritishDate1 date = new BritishDate1();
```

The type of the variable `date` declared in the above statement is `BritishDate1`. Then, is the following statement valid, and what is its implication?

```
Date1 date = new BritishDate1();
```

To test the above statement, the class `TestBritishDate1` is modified to `TestBritishDate2` with the above statement as shown in Figure 7.56.

```
// Definition of class TestBritishDate2
public class TestBritishDate2 {

    // Main executive method
    public static void main(String args[]) {
        Date1 date = new BritishDate1();
        date.setDay(3);
        date.setMonth(4);
        date.setYear(2003);
        date.print();
    }
}
```

Figure 7.56 TestBritishDate2.java

Is the above class definition valid? If it is valid, what is its output?

For the first statement in the `main()` method,

```
Date1 date = new BritishDate1();
```

we know that it first creates a `BritishDate1` object. However, is it possible to assign the reference of the `BritishDate1` object to the variable `date` of the type `Date1`?

In *Unit 3*, you learned that a variable declaration, say,

```
Date1 date;
```

declares a variable `date` of type `Date1`. Since the type `Date1` is not a primitive type, the variable `date` can store the reference of a `Date1` object. You also learned earlier in this unit that the `BritishDate1` class extends the `Date1` class, which means that `BritishDate1` 'is a' `Date1`. Therefore, a `BritishDate1` object can be considered a `Date1` object, and its reference can be stored by a variable of type `Date1`.

To make the idea even more explicit, let's remind ourselves of the little-box concept we came across in *Unit 3*. We said that a variable of non-primitive type could be considered a little box that stores a message to locate a real-world object in your home. For example, you remember that a little box named `counter` stores the message to find a ticket counter. When you need to use the ticket counter, you read the message in the little box and locate the object. You would not be surprised if the ticket counter you found is digital, because a digital ticket counter is a ticket counter. More exactly, a digital ticket counter is a specific type of a general ticket counter, and you can certainly use it like a normal ticket counter. This illustrates that a little box for a general type can store a message to find a specific type real-world object.

From another perspective, a subclass inherits all members (variables and methods) from its parent class. Therefore, whatever messages a superclass object can receive to perform corresponding behaviours, its subclass object can also accept them. For example, the second statement in the `main()` method of `TestBritishDate2` class

```
date.setDay(3);
```

sends a message `setDay` with supplementary data `3` to the object that is referred by the variable `date`. We know that the object that receives the message is a `BritishDate1` object. It inherits the `setDay()` method from the `Date1` class, and the method to be executed is the `setDay()` method defined in `Date1` class.

Similarly, the subsequent two statements,

```
date.setMonth(4);
date.setYear(2003);
```

send messages `setMonth` and `setYear` to the `BritishDate1` referred by the variable `date`, and the object executes the `setMonth()` and `setYear()` methods accordingly. A `BritishDate1` object can be visualized as shown in Figure 7.57.

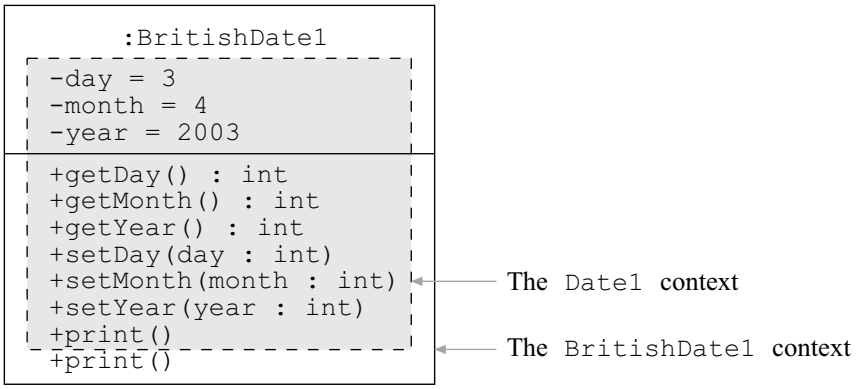


Figure 7.57 A `BritishDate1` object

The `BritishDate1` class is a subclass of the superclass `Date1`. Therefore, the `BritishDate1` class inherits all methods and attributes from the superclass `Date1` and a `BritishDate1` object has attributes and methods defined in both the superclass and the subclass. For illustration, the `BritishDate1` object in Figure 7.57 is partitioned into `Date1` context and `BritishDate1` context. The `Date1` context refers to the members defined in the `Date1` class, and the `BritishDate1` context refers to the members defined in the `BritishDate1` class.

The last statement in the `main()` method,

```
date.print();
```

sends a message `print` to the `BritishDate1` object via the variable `date`. Then, the `BritishDate1` object performs its `print()` method. As the `BritishDate1` object defines a `print()` method that overrides the one defined in the `Date1` class, the `print()` method to be executed is the one in the `BritishDate1` context and hence the `print()` method defined in the `BritishDate1` class. Therefore, although the variable `date` is of type `Date1`, the `print()` method to be executed is determined by the object itself. As a result, the method to be executed is the one defined in the `BritishDate1` class, and the message shown on the screen is:

```
03/04/2003
```

The classes `Date1` and `BritishDate1` illustrate two important facts:

1 A variable of type class `T` can store a reference to an object of a subclass of `T`.

2 The method to be executed is determined by the real type of the object rather than by the type of the variable that stores the reference of the object.

As a method to be executed or invoked is determined while the software is executing, it is known as the *late binding* or *virtual method invocation*. When the compiler software compiles a statement, say,

```
var.method();
```

it validates whether the type of the variable `var` can send the message `method` with the specified parameter list to the object to which it refers. When the statement is executed at runtime, the message `method` and the parameter values, if any, are sent to the object referred by the variable `var` so that the object can perform a corresponding method. The method to be executed is determined by the object itself.

A dictionary meaning of the word 'polymorphism' is *the ability of an entity to exist in different forms*. In the object-oriented paradigm, the word implies the ability of an object to behave differently on receiving the same message. Every true object-oriented programming language features polymorphism.

Furthermore, we said that a variable of class `T` can store an object reference of class `T` or any subclass of class `T`. Therefore, a variable of type `Date1` can store an object reference of class `Date1`, `BritishDate1` or `AmericanDate1`, which means that the statements in the following program segment are valid:

```
Date1 date;                 // statement 1
date = new BritishDate1();  // statement 2
date.setDay(1);             // statement 3
date.setMonth(1);           // statement 4
date.setYear(2003);         // statement 5
date.print();               // statement 6
date = new AmericanDate1(); // statement 7
date.setDay(31);            // statement 8
date.setMonth(12);          // statement 9
date.setYear(2003);         // statement 10
date.print();               // statement 11
```

The first statement declares a variable `date` of type `Date1`. The second statement creates a `BritishDate1` object, and the object reference is assigned to the variable `date.` It is valid because the `BritishDate1` class is a subclass of the `Date1` class, and a `BritishDate1` object can be treated as a `Date1` object. The third to fifth statements send messages `setDay`, `setMonth` and `setYear` to the `BritishDate1` object via the variable `date` of type `Date1`. Afterwards, the sixth statement sends the message `print` to the object referred by the variable `date`. The `BritishDate1` object performs its `print` behaviour according to the definition of the `BritishDate1` class and prints the date in the `dd/mm/yyyy` format.

Afterwards, the seventh statement creates an `AmericanDate1` object and assigns the reference of the created `AmericanDate1` object to the variable `date`. It is valid because the `AmericanDate1` class is also a subclass of the `Date1` class. (The reference to the `BritishDate1` object is lost, and the object will be removed by the JVM automatically.)

The program segment, the eighth to tenth statements, sends messages `setDay`, `setMonth` and `setYear` with a supplementary date for setting the attribute of the `AmericanDate1` object via the variable `date`. Finally, the `print` message is sent again to the object referred by the variable `date`. For the time being, the variable `date` is referring to an `AmericanDate1` object. Therefore, the `AmericanDate1` object performs its `print` behaviour in its own way — the `print()` method defined in the `AmericanDate1` class — and prints the date in the `m/d/yyyy` format.

You can see that with the same variable, the method to be executed depends on the actual class of the object. In other words, a variable can refer to objects of many forms or types that can behave differently. This is the way the Java programming language implements the idea of polymorphism.

Example: more information about `date`

To illustrate how polymorphism can be a powerful tool in object-oriented programming, a class `DateHandler1` is defined as shown in Figure 7.58.

```
// Definition of class DateHandler1
public class DateHandler1 {
    // Attributes

    // An array for storing all month names
    private static final String[] MONTH_NAMES = {
        "January", "February", "March", "April",
        "May", "June", "July", "August",
        "September", "October", "November", "December"
    };

    // Methods
    // Show the details of a Date1 object
   public void saySomethingAboutDate(Date1 date) {
      System.out.print("The date (");
      date.print();
      System.out.print(") means ");
      System.out.print(MONTH_NAMES[date.getMonth() - 1]);
      System.out.print(" "  + date.getDay());
      System.out.println(", " + date.getYear() + ".");
   }
}
```

Figure 7.58 DateHandler1.java

The `DateHandler1` class defines a method `saySomethingAboutDate()` that can accept a reference of a `Date1` object. When the method is called with supplementary data of a reference of a `Date1` object, it shows a statement about it on the screen. To test the `DateHandler1` class, a `TestDateHandler1` class is defined in Figure 7.59.

```
// Definition of class TestDateHandler1
public class TestDateHandler1 {

   // Main executive method
   public static void main(String args[]) {
     // Create a DateHandler1 object and is referred by variable
     // handler
     DateHandler1 handler = new DateHandler1();

     // Create a Date1 object
     Date1 date1 = new Date1();
     date1.setDay(1);
     date1.setMonth(2);
     date1.setYear(2001);
     // Call the DateHandler1 object with the Date1 object referred
     // by variable date1
     handler.saySomethingAboutDate(date1);

     // Create a BritishDate1 object
     BritishDate1 date2 = new BritishDate1();
     date2.setDay(3);
     date2.setMonth(4);
     date2.setYear(2002);
     // Call the DateHandler1 object with the BritishDate1 object
     // referred by variable date2
       handler.saySomethingAboutDate(date2);

     // Create an AmericanDate1 object
     AmericanDate1 date3 = new AmericanDate1();
     date3.setDay(5);
     date3.setMonth(6);
     date3.setYear(2003);
     // Call the DateHandler1 object with the AmericanDate1 object
     // referred by variable date3
       handler.saySomethingAboutDate(date3);
   }
}
```

Figure 7.59 TestDateHandler1.java

The `main()` method of the `TestDateHandler1` class first creates a `DateHandler1` object that is referred by the variable `handler`. Afterwards, a `Date1` object is created that is referred by the variable `date1`. After setting the object attributes `day`, `month` and `year` by calling the corresponding methods, the `saySomethingAboutDate()` method is called and is supplied the content of the variable `date1`, which is the reference of the `Date1` object. Therefore, when the `DateHandler1` object is performing its `saySomethingAboutDate()` method, the parameter `date` is referring to the `Date1` object that is referred by the variable `date1` in the `main()` method as shown in Figure 7.60.

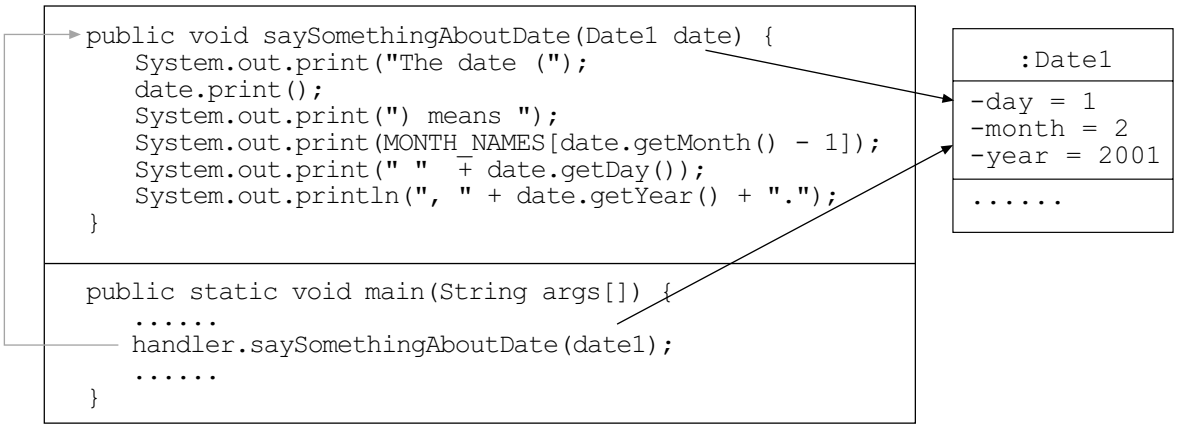


Figure 7.60 Calling the saySomethingAboutDate() method with a Date object

When the `saySomethingAboutDate()` method is executing, the message

```
The date (
```

is shown on the screen first. Afterwards, a message `print` is sent to the object referred by the parameter `date`, which is a `Date1` object. Therefore, the `Date1` object executes the `print()` method that is defined in the `Date1` class. The following output is shown on the screen:

```
20010201
```

Afterwards, the `getDay()`, `getMonth()` and `getYear()` methods of the `Date1` object referred by the parameter `date` is called for getting the object attribute values, and the date is shown in a descriptive way. As a whole, the following message is shown by calling the `saySomethingAboutDate()` method with the reference to the `Date1` object referred by the variable `date1` in the `main()` method:

```
The date (20010201) means February 1, 2001.
```

After the above message is shown on the screen, the flow of control is returned to the `main()` method of the `TestDateHandler1` class. The following program segment,

```
// Create a BritishDate1 object
BritishDate1 date2 = new BritishDate1();
date2.setDay(3);
date2.setMonth(4);
date2.setYear(2002);
// Call the DateHandler1 object with the BritishDate1 object
// referred by variable date2
   handler.saySomethingAboutDate(date2);
```

then creates a `BritishDate1` object that is referred by variable `date2`. After setting the `BritishDate1` object attributes with the setter methods, the `DateHandler1` object referred by the variable `handler` is sent a `saySomethingAboutDate` message with supplementary data from the reference of the `BritishDate1` object

stored in the variable `date2`. Is there any problem with supplying the `saySomethingAboutDate()` method from a `BritishDate1` object if its definition needs a `Date1` object?

The `saySomethingAboutDate()` method needs a `Date1` object; as `BritishDate1` is a `Date1` (because of the 'is a' relationship between the classes `BritishDate1` and `Date1`), the `BritishDate1` object is treated as if it is a `Date1` object, and the `saySomethingAboutDate()` method therefore accepts it. From another perspective, the `saySomethingAboutDate()` method needs supplementary data as the reference of an object having all attributes and methods defined by the `Date1` class. As the `BritishDate1` class is a subclass of the `Date1` class, it inherits all attributes and methods that are defined by the `Date1` class. Therefore, it is acceptable for the `saySomethingAboutDate()` method to accept a `BritishDate1` object. This illustrates that a method that expects and handles an object of a superclass (a general type) can accept an object of all its subclasses.

When the `saySomethingAboutDate()` method is executed, its parameter `date` is assigned the contents of the variable `date2` in the `main()` method. Therefore, when the `saySomethingAboutDate()` method is executing, the parameter `date` is referring to the `BritishDate1` object, which is the same one referred by the `date2` variable in the `main()` method as shown in Figure 7.61.

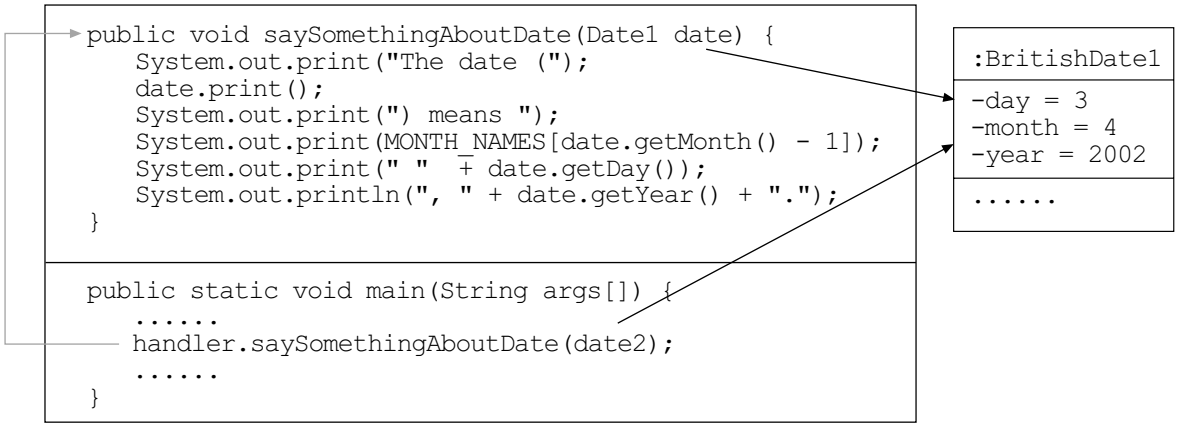


Figure 7.61 Calling the `saySomethingAboutDate()` method with a `BritishDate1` object

After the message

```
The date (
```

is shown on the screen, a message `print` is sent to the object referred by the parameter `date`, which is a `BritishDate1` object. Therefore, it executes the `print()` method that is defined in the `BritishDate1` class. The message to be shown on the screen is:

```
03/04/2002
```

The remaining statements in the `saySomethingAboutDate()` method work in the same way as in the previous method execution. Therefore, the following message is shown after the second method call to the `saySomethingAboutDate()` with a `BritishDate1` object:

```
The date (03/04/2002) means April 3, 2002.
```

The flow of control returns to the `main()` method of the `TestDateHandler1` class, and the following program segment is executed:

```
// Create an AmericanDate1 object
AmericanDate1 date3 = new AmericanDate1();
date3.setDay(5);
date3.setMonth(6);
date3.setYear(2003);
// Call the DateHandler1 object with the AmericanDate1 object
// referred by variable date3
   handler.saySomethingAboutDate(date3);
```

The program segment creates an `AmericanDate1` object that is referred by the variable `date3`. After setting the object attribute values, the `DateHandler1` object receives the message `saySomethingAboutDate` with the reference of an `AmericanDate1` object. Similar to the `BritishDate1` class, the `AmericanDate1` class is a subclass of the `Date1` class. An `AmericanDate1` object has all attributes and methods defined by the `Date1` class, and an `AmericanDate1` object is treated as if it were a `Date1` object. Therefore, it is possible to supply the `saySomethingAboutDate()` method with a reference to an `AmericanDate1` object as shown in Figure 7.62.

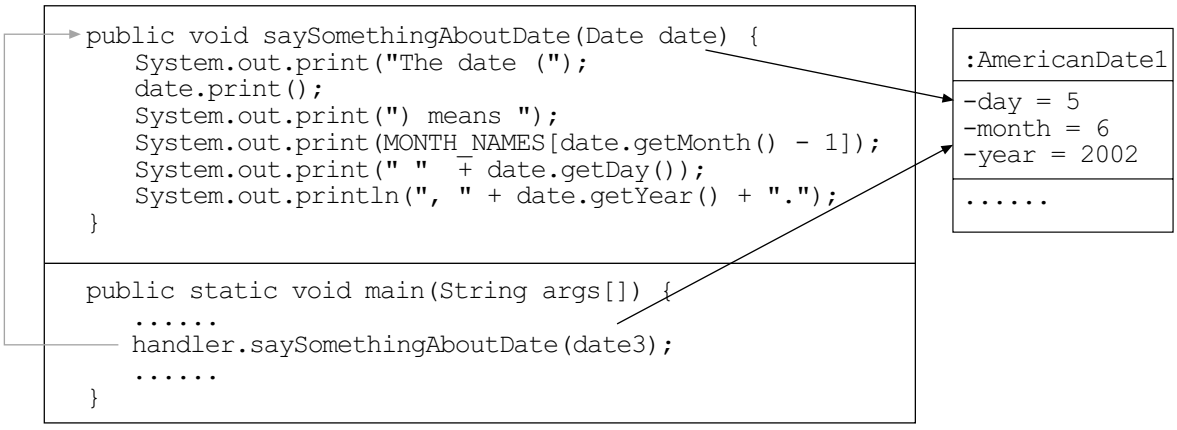


Figure 7.62 Calling the saySomethingAboutDate() method with an AmericanDate1 object

The method execution of `saySomethingAboutDate()` method with the `AmericanDate1` object shows the following message:

```
The date (6/5/2003) means June 5, 2003.
```

You can see that the output `6/5/2003` is shown because the `AmericanDate1` object is sent a message `print` in the `saySomethingAboutDate()` method, and it executes the `print()` method that is defined in the `AmericanDate1` class.

As a result, the execution of the `TestDateHandler1` class prints out the following message on the screen:

```
The date (20010201) means February 1, 2001.
The date (03/04/2002) means April 3, 2002.
The date (6/5/2003) means June 5, 2003.
```

The method `saySomethingAboutDate()` is written to handle a `Date1` object. Polymorphism can handle the general type `Date1` and all of its specific types such as `BritishDate1` and `AmericanDate1`.

As you can see, a variable of a superclass type can refer to objects of the class and its subclasses. If it is necessary to perform particular operations for particular subclass type, we can use the `instanceof` operator and casting operations. Please refer to Appendix H for further details.

In the following reading, you will find more examples and advantages of polymorphism.

Reading

King, section 11.4, pp. 453–57

Please use the following self-test to test the applications of polymorphism.

Self-test 7.8

Based on the definition of the class `ChineseDate1` written in Self-test 7.7, modify the `main()` method of the `TestDateHandler1` class by adding a program segment so that the following messages are shown on the screen.

```
The date (20010201) means February 1, 2001.
The date (03-04-2002) means April 3, 2002.
The date (06-05-2003) means June 5, 2003.
The date (2004.8.7) means August 7, 2004.
```

Self-test 7.8 demonstrates the beauty of polymorphism in that the `DateHandler1` needs no modification to handle a `ChineseDate1` object, but the date shown in the output is in the specific format defined in the `print()` method of the `ChineseDate1` class. That is, an existing, workable and thoroughly tested class definition, the `DateHandler1` class in this case, can be reused to handle new specific

types (such as the `ChineseDate1` class) of the general type (the `Date1` class). The implication is that the software can be easily reused and its reliability is guaranteed.

The `super` keyword — calling parent class constructors

Up to now, the classes `Date1` and their subclasses are defined without constructors. The reason is that constructors are handled differently with respect to inheritance. A subclass does not inherit constructors from its parent class, which means that the way to create an object of a parent class is not necessarily the same as the way to create its subclasses. Therefore, we have to study the issues of constructors separately.

First of all, let's investigate the issue of the constructor for the `Date1` class. According to its class definition, the `Date1` class defines no constructor. Therefore, a default constructor is added implicitly to the class definition. As a result, there is an implicit constructor for the `Date1` class as if the class definition is:

```
public class Date1 {
    // Attributes
    ......

    // Constructor
    public Date1() {
    }

    // Methods
    ......
}
```

The implicit default constructor enables a `Date1` object to be created with the statement:

```
   new Date1()
```

For a constructor without the `this(`*parameter-value-list*`)` statement as the first statement, an implicit statement `super()` is added before all statements in the constructor. That is, the constructor of the `Date1` class is more exactly like:

```
    // Constructor
    public Date1() {
        super();
    }
```

The effect of the statement `super(`*parameter-value-list*`)` calls the constructor of the immediate superclass with a parameter list that matches *parameter-value-list*, so that the superclass constructor can initialize the superclass context. If the superclass constructors are overloaded, the parameter value list in the `super()` statement determines the superclass constructor to be executed. During the creation process of a `Date1` object, as the superclass of the `Date1` class is the `Object` class, the implicit statement `super()` in the implicit default constructor of the

`Date1` class will call the constructor with an empty parameter list of the `Object` class before any statement in the `Date1` class constructor is executed. The constructor of the `Object` class performs nothing, and the flow of control returns immediately to the `Date1` constructor. The statements in the `Date1` constructor will then be executed.

To define a constructor for the date-related classes, we should start from the superclass. A common superclass `Date2` with a constructor is defined as shown in Figure 7.63.

```
// definition of the class Date2
public class Date2 {
    // Attributes
    private int day;        // the day
    private int month;      // the month
    private int year;       // the year

    // Constructor
    public Date2(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    // Getter methods

    // get the day
    public int getDay() {
        return day;
    }

    // get the month
    public int getMonth() {
        return month;
    }

    // get the year
    public int getYear() {
        return year;
    }

    // Setter methods

    // set the day
    public void setDay(int day) {
        this.day = day;
    }

    // set the month
    public void setMonth(int month) {
        this.month = month;
    }

    // set the year
    public void setYear(int year) {
```

```
        this.year = year;
    }

    public void print() {
        System.out.print(year);
        System.out.print((month < 10) ? "0" + month : "" + month);
        System.out.print((day < 10) ? "0" + day : "" + day);
    }
}
```

Figure 7.63 Date2.java

The constructor of the `Date2` class simply assigns the values stored in the parameters `day`, `month` and `year` to the object attributes `day`, `month` and `year`. The keyword `this` is necessary, as it identifies the object attributes from the parameters in the statements. Furthermore, as there is no `this()` statement as the first statement of the constructor, an implicit `super()` statement is added. Therefore, you can imagine that the constructor for the `Date2` class is actually:

```
public Date2(int day, int month, int year) {
    super();
    this.day = day;
    this.month = month;
    this.year = year;
}
```

As the `super()` statement will call the constructor of the `Object` class that performs nothing, this `super()` statement is usually omitted for most cases — as in the `Date2` class definition shown in Figure 7.63. Furthermore, the `Date2` class defines its own constructor, and no implicit default constructor is added to the class. This means that it is no longer possible to create a `Date2` object by a statement that looks like

```
new Date2()
```

and the way to create a `Date2` object must be in the form that looks like

```
new Date2(1, 2, 2003)
```

which creates a `Date2` object with initial object attributes `day`, `month` and `year` of values `1`, `2` and `2003` respectively.

If a class `BritishDate2` were defined to be the subclass of the `Date2` class without a constructor

```
public class BritishDate2 extends Date2 {
    // Attributes
    ......

    // Methods
    ......
}
```

an implicit constructor with an implicit `super()` statement is added to the class. That is, the class can be considered to be:

```
public class BritishDate2 extends Date2 {
    // Attributes
    ......

    // Implicit default Constructor
    public BritishDate2() {
        super();
    }

    // Methods
    ......
}
```

As there is an implicit `super()` statement in the `BritishDate2` constructor, it is expected that a `Date2` class constructor with an empty parameter list will be called during the creation process of a `BritishDate2` object. However, the `Date2` class defines no constructor with an empty parameter list, so the compiler software gives a compile-time error.

The parameter value list of the `super()` statement determines the superclass constructor to be called, so that we can explicitly provide our own `super()` statement with a suitable parameter value list to determine the desired superclass constructor to be called during the creation process. Please notice that the explicit `super()` statement must be the first statement in any subclass constructor.

For the `BritishDate2` class, a possible constructor can be written as shown in Figure 7.64.

```
// Definition of class BritishDate2
public class BritishDate2 extends Date2 {
    // Constructor
    public BritishDate2(int day, int month, int year) {
        super(day, month, year);
    }
    // Override the print() method defined in the Date1 class
    // and show the date in dd/mm/yyyy format
    public void print() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in dd/mm/yyyy format
        System.out.print((day < 10) ? "0" + day : "" + day);
        System.out.print("/");
        System.out.print((month < 10) ? "0" + month : "" + month);
        System.out.print("/");
        System.out.print(year);
    }
}
```

Figure 7.64 BritishDate2.java

To create a `BritishDate2` object, you can now use a statement like:

```
new BritishDate2(1, 2, 2003)
```

The statement first instructs the JVM to allocate a memory block for the object, and all object attributes defined in the class and all its superclasses, `Date2` and `Object`, are initialized to the default values. You learned earlier in the unit that this is known as implicit initialization.

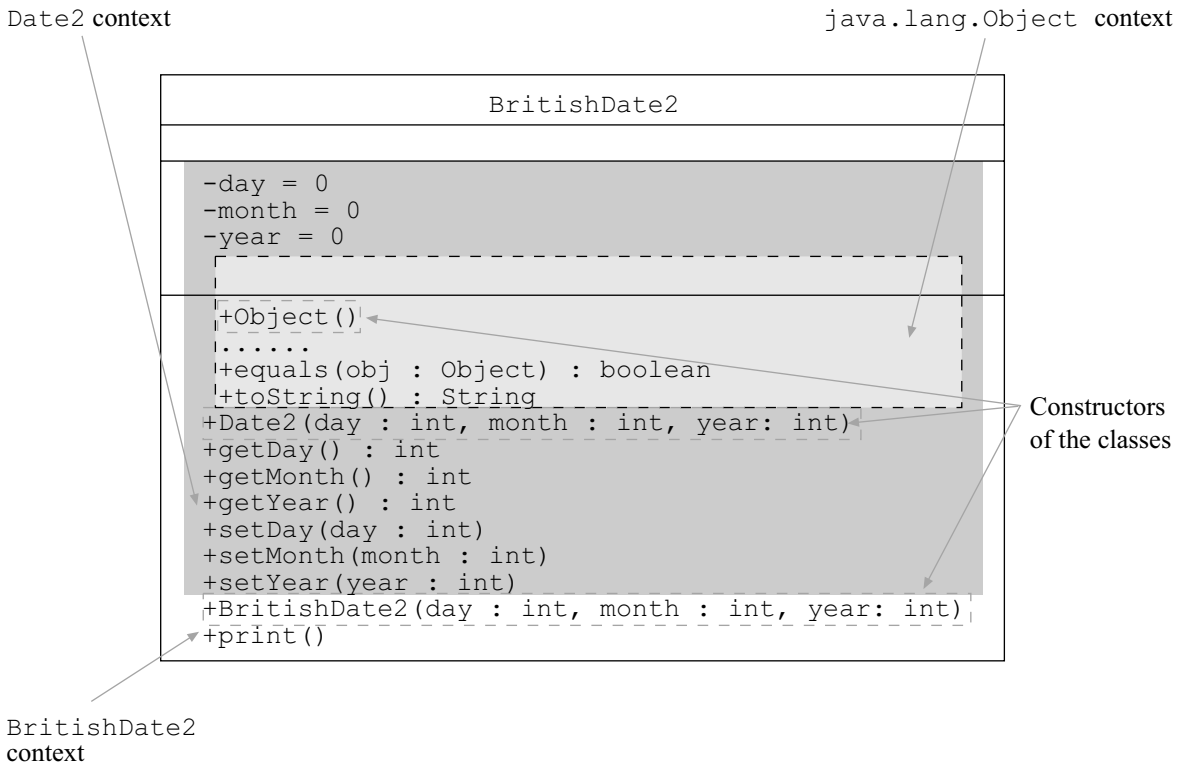


Figure 7.65 The BritishDate2 object after implicit initialization

The `BritishDate2` constructor defines an explicit `super()` statement with a parameter list of three `int` values. Therefore, the constructor of its superclass, the class `Date2`, is executed with the parameters supplied, as shown in Figure 7.66.

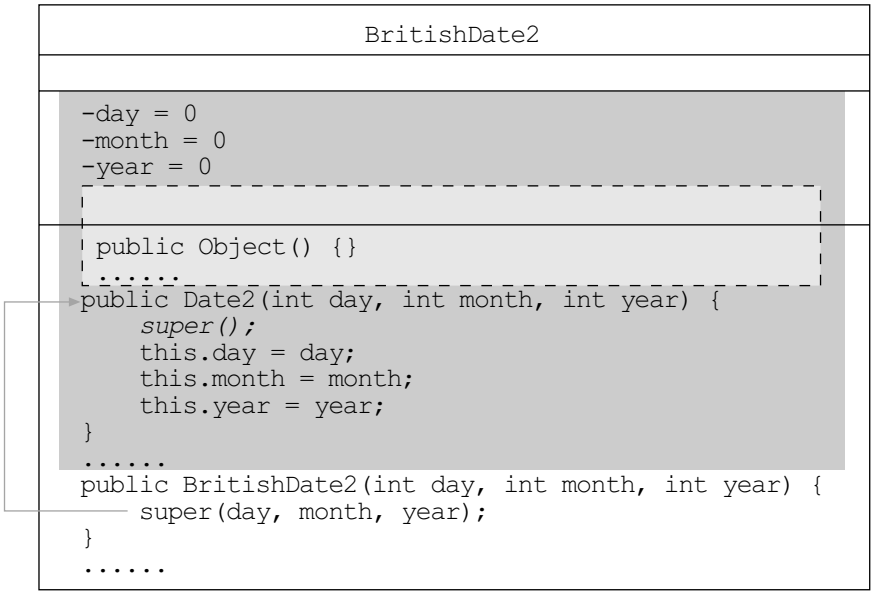


Figure 7.66 The super() statement in the BritishDate2 constructor calls Date2 class constructor

(The `super()` statement in the `Date2` constructor is italicized to indicate it is an implicit statement rather than actually written in the class definition.)

Then, the constructor of the `Date2` class is executed. The first statement is the implicit `super()` statement, which calls the constructor of the `Object` class as shown in Figure 7.67.

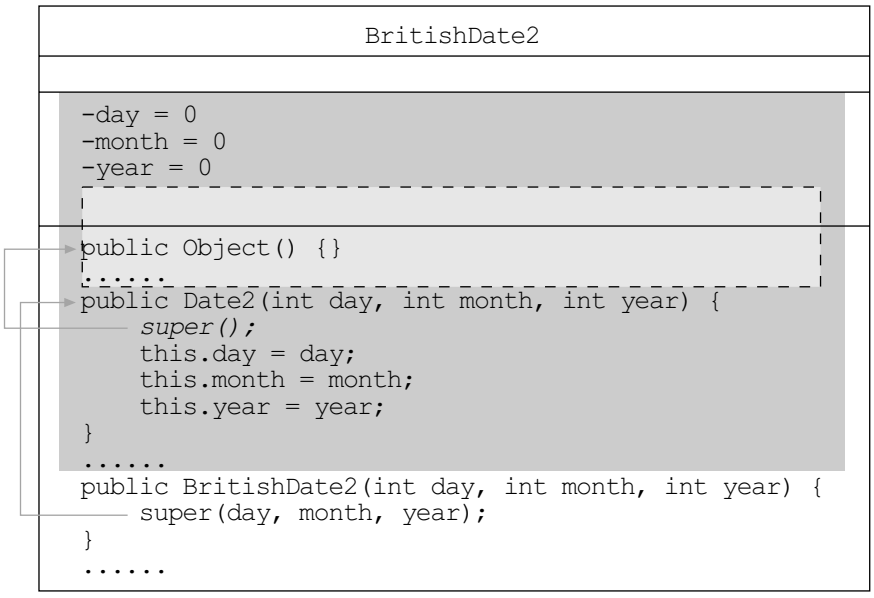


Figure 7.67 The super() statement in the Date2 constructor calls the Object class constructor

As the `Object` class constructor performs nothing, the flow of control immediately returns to the `Date2` constructor. The following three statements are executed to assign the parameter values to the object attributes:

```
this.day = day;
this.month = month;
this.year = year;
```

The `Date2` constructor is terminated, and the flow of control is returned to the `BritishDate2` constructor. The `BritishDate2` object becomes as shown in Figure 7.68.

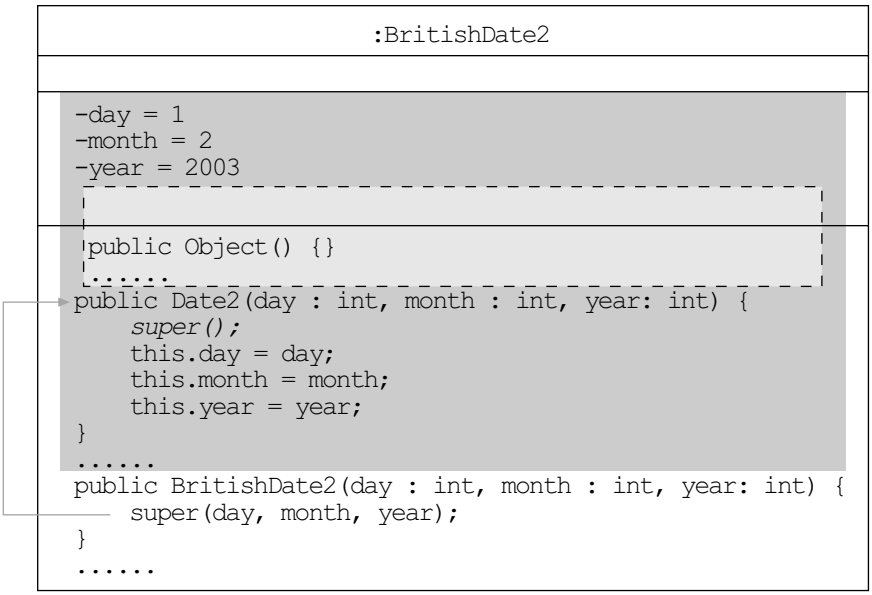


Figure 7.68 The BritishDate2 object after the Date2 constructor is executed

In the `BritishDate2` class constructor, there is no statement after the `super()` statement, and the constructor terminates. This is the end of the creation process of a `BritishDate2` object. The creation process of the `BritishDate2` object is completed. The object is visualized as shown in Figure 7.69.

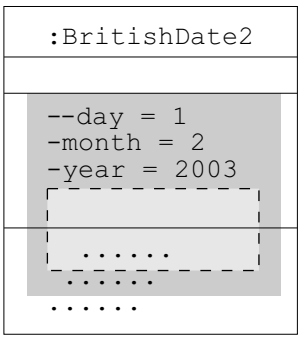


Figure 7.69 A BritishDate2 object after the creation process

Similarly, it is possible to define the `AmericanDate2` class as a subclass of the `Date2` class. Please use the following self-test to try adding a constructor to a subclass.

Self-test 7.9

Define the `AmericanDate2` class as a subclass of the `Date2` class, so that the constructor accepts three parameter values of `day`, `month` and `year`. The `AmericanDate2` class should not assign the parameter values to the object attributes by calling the setter methods; a `super()` statement should be used instead.

The `super` keyword — accessing overridden members of the parent class

Besides calling the constructor of the superclass, the keyword `super` can be used to refer to an attribute or a method defined in the superclass, especially those overridden in the subclass. For example, for the classes `Staff2` and `Manager2` discussed earlier in this unit, we can derive the `Staff3` class by adding a constructor and a `toString()` method as shown in Figure 7.70.

```
// Definition of class Staff3
public class Staff3 {
    // Attributes
    private String name; // The staff name
    private String number; // The staff number
    private double basicSalary; // The basic salary

    // Constructor
    public Staff3(String name, String number, double basicSalary) {
        this.name = name;
        this.number = number;
        this.basicSalary = basicSalary;
    }

    // Behaviours

    // Get the staff name
    public String getName() {
        return name;
    }

    // Get the staff number
    public String getNumber() {
        return number;
    }

    // Get the basic salary
    public double getBasicSalary() {
        return basicSalary;
    }

    // Set the name of the staff
    public void setName(String theName) {
        name = theName;
```

```
    }

    // Set the number of the staff
    public void setNumber(String theNumber) {
        number = theNumber;
    }

    // Set the salary of the staff
    public void setBasicSalary(double theBasicSalary) {
        basicSalary = theBasicSalary;
    }

    // Return a textual representation of the object
    public String toString() {
        return
            "Name=" + name +
            ";Number=" + number +
            ";Basic salary=" + basicSalary;
    }
}
```

Figure 7.70 Staff3.java

With the definition of `Staff3` class, the following program segment

```
Staff3 staff = new Staff3("Peter", "0001", 10000.0);
System.out.println(staff);
```

will first create a `Staff3` object with values of object attributes `name`, `number` and `basicSalary`, which are `"Peter"`, `"0001"` and `10000.0` respectively. For the second statement that calls the `println()` method, supplying a non-primitive reference other than types `String` and `char[]` will call the following overloaded `println()` method

```
println(java.lang.Object x)
```

and the `toString()` method of the object supplied to the method will be called implicitly.

The above `println()` can accept any object reference, because all classes are subclasses of the class `Object`. The above statement is practically equivalent to:

```
System.out.println(staff.toString());
```

Please refer to Appendix B for a detailed discussion on the `Object` class.

The `toString()` method returns the reference of a `String` object with the textual representation of the contents of the object. Therefore, the above program segment will show the following output on the screen:

```
Name=Peter;Number=0001;Basic salary=10000.0
```

To define the subclass `Manager3`, remember it is an extension of the `Staff3` class with an extra attribute `commission` and the corresponding setter/getter method. As its superclass `Staff3` has no constructor with an empty parameter list, it is necessary to provide an *explicit* `super()` statement in the constructor, such as:

```
public Manager3(String name,
                String number,
                double basicSalary,
                double commission) {
    super(name, number, basicSalary);
    this.commission = commission;
}
```

The constructor first calls the superclass constructor by using the `super()` statement and supplies the values for parameters `name`, `number` and `basicSalary` so that the constructor defined in the `Staff3` can initialize the corresponding attributes accordingly. Afterwards, when the flow of control returns to the `Manager3` constructor, the statement

```
    this.commission = commission;
```

assigns the value of the parameter `commission` to the object attribute `commission`.

For the `toString()` method of the `Manager3` class, it is expected that a `String` object would be returned in a form that looks like:

```
Name=Peter;Number=0001;Basic
salary=10000.0;Commission=5000.0
```

A possible way to define the `toString()` method is:

```
public String toString() {
    return
        "Name=" + getName() +
        ";Number=" + getNumber() +
        ";Basic salary=" + getBasicSalary() +
        ";Commission=" + commission;
}
```

As the attributes `name`, `number` and `basicSalary` defined in the `Staff3` class are marked `private`, the subclass `Manager3` cannot access them directly but must access them indirectly via the getter methods.

You can see that part of the statement in the `Staff3` class `toString()` is similar to the above `Manager3` class `toString()` method. It is preferable to use the `toString()` method defined in the `Staff3` class to construct the `String` object in the `Manager3` class `toString()` method. Then, you can use the keyword `super` to refer to the `toString()` method defined in the superclass `Staff3` as follows:

```
public String toString() {
   return super.toString() + ";Commission=" +
   commission;
}
```

In the definition of the `Manager3` class, a method call to the `toString()` method without any prefix or prefixed by '`this.`' refers to the `toString()` method defined in `Manager3`. To execute a method inherited from the superclass but which is usually overridden in the subclass, it is necessary to use a prefix '`super.`' as in the statement `super.toString()`. In the above `toString()` method, if the prefix '`super.`' were missing, the `toString()` method defined in the `Manager3` class would have been called recursively (infinitely), and the program would have terminated with runtime error.

The complete definition of the `Manager3` class is shown in Figure 7.71.

```
// Definition of class Manager3
public class Manager3 extends Staff3 {
    // Attributes
    private double commission; // The commission

    // Constructor
    public Manager3(String name,
                    String number,
                    double basicSalary,
                    double commission) {
        super(name, number, basicSalary);
        this.commission = commission;
    }

    // Behaviours

    // Get the commission
    public double getCommission() {
        return commission;
    }

    // Set the commission
    public void setCommission(double theCommission) {
        commission = theCommission;
    }

    // Return the textual representation of the object
    public String toString() {
        return super.toString() + ";Commission=" + commission;
    }
}
```

Figure 7.71 Manager3.java

To test the classes `Staff3` and `Manager3`, a class `TestSuper` is written in Figure 7.72.

```
// Definition of class TestSuper
public class TestSuper {

    // Main executive method
    public static void main(String args[]) {
        // Create a Staff3 object and a Manager3 object
        Staff3 staff1 = new Staff3("Peter", "0001", 10000.0);
        Staff3 staff2 = new Manager3("John", "0002", 20000.0,
                                     10000.0);

        // Show the objects
        System.out.println(staff1);
        System.out.println(staff2);
    }
}
```

Figure 7.72 TestSuper.java

The first two statements in the `main()` method of the `TestSuper` class create a `Staff3` object and a `Manager3` object that are referred by the variables `staff1` and `staff2` respectively. Since a `Manager3` object can be treated as a `Staff3` object, it is possible to use the variable `staff2` of the type `Staff3` to refer to the `Manager3` object.

After the first two statements, the scenario can be visualized in Figure 7.73.

Afterwards, the `System.out.println()` method is called, and the reference stored in the variable `staff1` is supplied as supplementary data. Then, the `toString()` method of the object referred by the variable `staff1` is implicitly called, which means a message `toString` is sent to the `Staff3` object. According to the definition of the class `Staff3`, the `toString()` method defined in the class overrides the one defined in the `Object` class. Therefore, the `toString()` method defined in the `Staff3` class is executed, and a `String` object involving the values of object attributes `name`, `number` and `basicSalary` is returned and is shown on the screen.

```
Name=Peter;Number=0001;Basic salary=10000.0
```

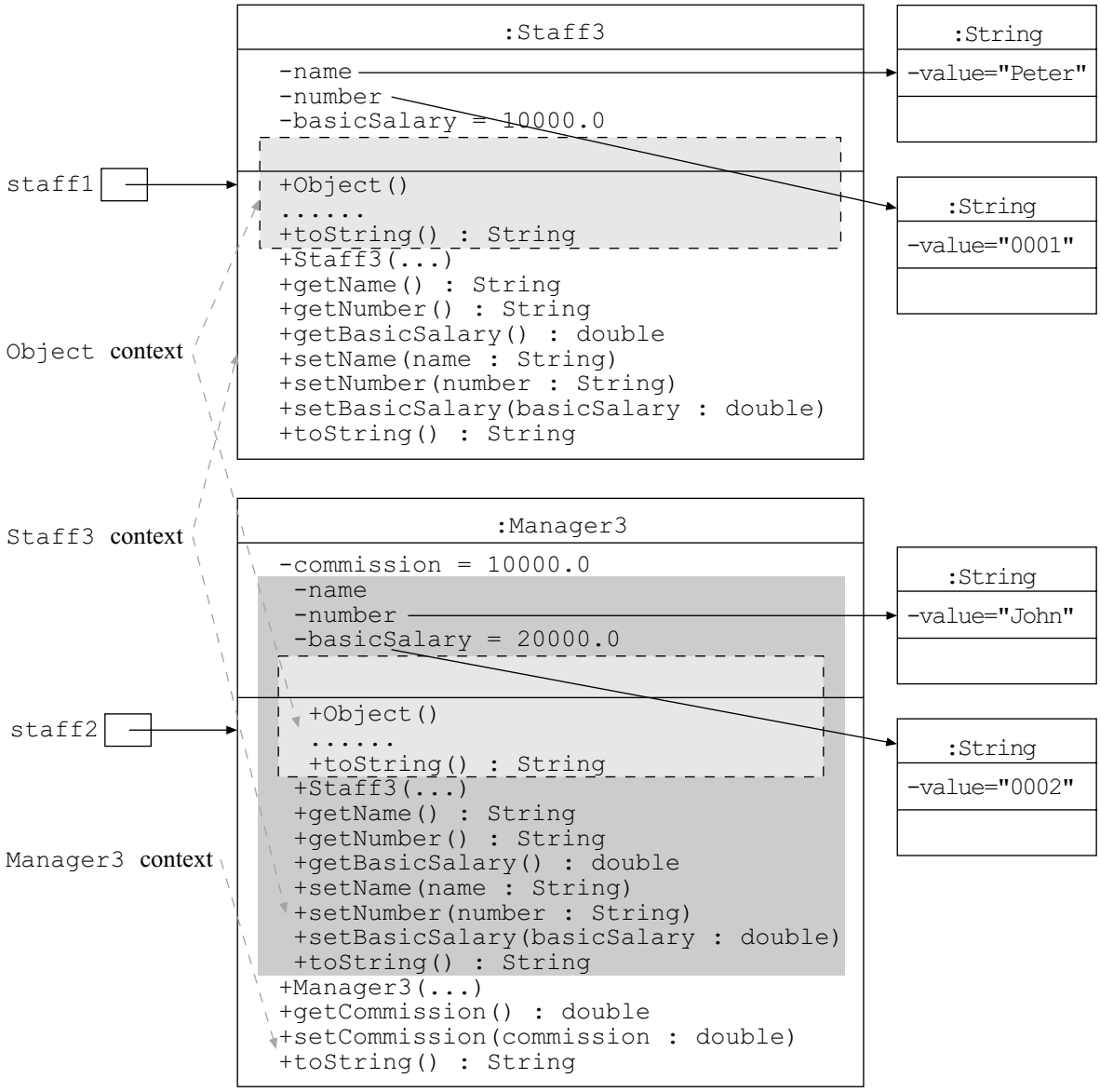


Figure 7.73 A Staff3 object and a Manager3 object

Finally, the `System.out.println()` method is called and the reference stored in the variable `staff2` is supplied as supplementary data. Like the previous statement, the `toString()` method of the object that is supplied to the `System.out.println()` is implicitly called, meaning a `toString` message is sent to the `Manager3` object. The `toString()` method defined in the `Manager3` class, which overrides the one defined in the `Staff3` class, is executed:

```
public String toString() {
  return super.toString() + ";Commission=" + commission;
 }
```

While executing the `toString()` method of the `Manager3` class, the statement `super.toString()` executes the `toString()` method that is inherited by the `Manager3` class, which is the `toString()` method defined in the `Staff3` class. Figure 7.74 shows the relationships.

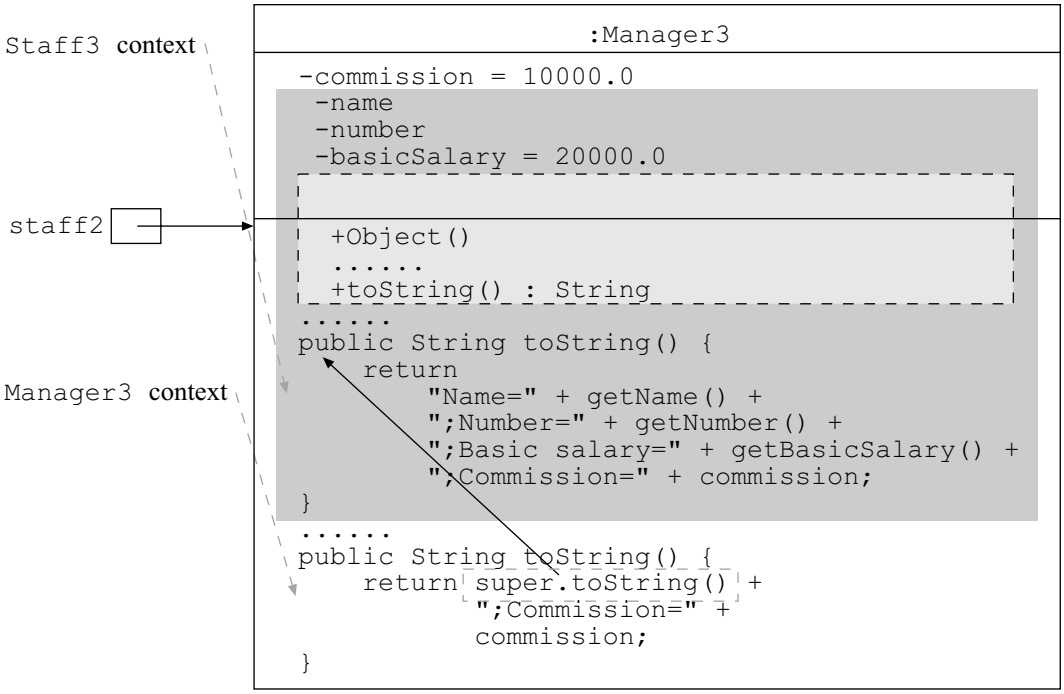


Figure 7.74 Calling the `toString()` method defined in the `Staff3` class from within the `toString()` method defined in `Manager3` class with `super.toString()`

The execution of the `toString()` method defined in the `Staff3` class returns a `String` object with contents

```
Name=John;Number=0002;Basic salary=20000.0
```

The flow of control returns to the `toString()` method defined in the `Manager3` class, and the `String` object returned by the statement `super.toString()` is concatenated with the string `";Commission="` and the value of the attribute `commission`. As a result, a `String` object with contents

```
Name=John;Number=0002;Basic
salary=20000.0;Commission=10000.0
```

is returned by the `toString()` method by the `Manager3` object and is then shown on the screen. Therefore, by compiling the classes and executing the `TestSuper` class, the following output is shown on the screen:

```
Name=Peter;Number=0001;Basic salary=10000.0
Name=John;Number=0002;Basic
salary=20000.0;Commission=10000.0
```

There is no way to call the `toString()` method defined in the `Object` class from within a method defined in the `Manager3` class, because its superclass `Staff3` defines the `toString()` method that overrides the one defined in the `Object` class. If the `Staff3` class defined no `toString()` method, the `toString()` method to be called by the statement `super.toString()` from within a method of

`Manager3` would have been the one defined in the `Object` class as shown in Figure 7.75.

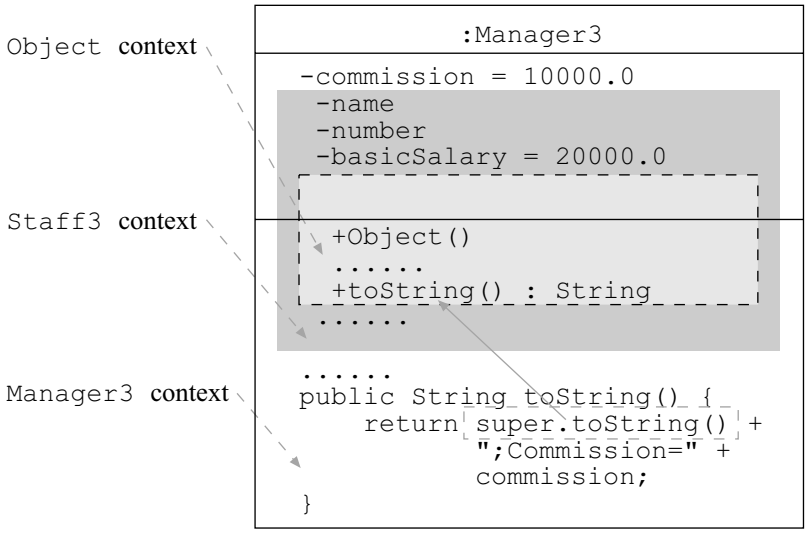


Figure 7.75 The `Object` class `toString()` method is called by `super.toString()` from within the method defined in the `Manager3` class, if the `Staff3` class does not define an overriding `toString()` method

An abstract blueprint — `abstract` class

You learned that a class definition defines the behaviours and attributes for a group of objects to be created at runtime. Sometimes, a programmer might find that he or she knows how an object must behave but has no idea how the object performs some of the behaviour. That is, for some methods, the programmer can figure out the method name and the parameter list, but the statements in the method body cannot be determined for the time being.

For example, an international software company is developing a tax calculation program that can be used by people in different countries for tax calculation. Among all the classes involved, it is determined that the software needs a class `TaxCalculator` with the method `findTax()` for tax calculation. However, the rules of tax calculations vary in different countries, and it is almost impossible for a programmer to incorporate the tax calculations for all countries in one method.

Inheritance in the object-oriented paradigm can be a powerful tool to handle such a problem. For example, it is possible to define a general `GeneralTaxCalculator` class with the method `findTax()`. Then, the regional office of the company in each country can write a specific subclass of the superclass `GeneralTaxCalculator` — for example `HongKongTaxCalculator` — that defines its own `findTax()` method. Then, by adding the class `HongKongTaxCalculator` to the software application, the software can be used in Hong Kong by creating a `HongKongTaxCalculator` object and used as if it is a `GeneralTaxCalculator` object, such as:

```
GeneralTaxCalculator calculator = new HongKongTaxCalculator();
......
calculator.findTax(...);
```

As a result, all statements other than the statement that creates the object can be kept unchanged because of polymorphism.

However, there is no general global tax calculation method, so it is unreasonable to define a `findTax()` method in the `GeneralTaxCalculator` class. However, if the class `GeneralTaxCalculator` does not define the `findTax()` method, the statement

```
calculator.findTax(...);
```

will cause a compilation error.

The Java programming language supports abstract classes and abstract methods to resolve the above problem. Briefly, an abstract class is written as a superclass that defines common attributes and concrete (or non-abstract) methods that its subclass must possess and some abstract

methods its subclasses must define. Please use the following reading to learn what an abstract class and an abstract method are, and their uses.

Reading

King, section 11.6, pp. 461–65

From the reading, you know that an abstract class enables you to model some conceptual ideas that are actually non-existent, e.g. there is no generic `Vehicle` or `Account` as mentioned in the reading. If something can be considered a vehicle, however, you are sure that it can start, move and stop. The ways it performs such operations are unknown until you really know what a vehicle it is, such as a car or a plane.

An abstract class defines a general type and what behaviours such a general type can perform (or the message types it can receive), and some of the behaviours can be undefined. This leaves room for its specific types (that is, the subclasses) to realize or define those behaviours. Therefore, an abstract class acts as a blueprint of its subclasses.

What are abstract methods and why do we need them?

From the programming point of view, an abstract method is a method definition marked `abstract`, and the method body, which includes the pair of curly braces, is replaced by a semi-colon. The general format is:

```
public abstract return-type method-name(parameter-list);
```

For example, the `abstract` methods defined in the `Shape` class mentioned in the previous reading are:

```
public abstract void draw(Graphics g);
public abstract int getHeight();
public abstract int getWidth();
```

For a generic `Shape` to be drawn on the screen, it must have attributes including coordinates and colour. However, we know nothing about the way it is drawn on the screen and how to determine its dimensions. The baseline is that we know a specific real or concrete `Shape` must have such behaviours. Therefore, these methods are defined to be `abstract` methods and are left to be defined in its specific types or subclasses.

From the reading, you learned that if a class defines one or more abstract methods, the class has to be marked `abstract` as well. This confirms that you are intentionally defining an abstract class. Furthermore, because abstract classes model a conceptual idea or a general type, it is not possible to create an object of an abstract class. Therefore, with the

definition of class `Shape` mentioned in the reading, the following statement

```
new Shape()
```

will cause a compile-time error.

The outcome is that if an object can be considered a `Shape` object, its actual class must not be `Shape` but a non-abstract subclass of the `Shape` class. The reason is that it is not possible to create an object of an abstract class. If an object is referred to by a variable of type `Shape`, it must be an object of a non-abstract subclass of the `Shape` class. Moreover, all abstract methods must be defined in the non-abstract subclasses. The reason is that when an object receives a message that is defined abstract in the abstract superclass, it needs to execute the corresponding method defined by the class (a concrete subclass of the abstract superclass) of the object.

With respect to the `Shape` class, the reading mentions two specific types (subclasses), `Circle` and `Rectangle`. The two subclasses inherit all attributes and methods defined in the superclass `Shape`, excluding the constructor. Therefore, the subclasses only need to define a constructor and all `abstract` methods defined in the `Shape` class. Then, all methods defined in the two subclasses are non-`abstract`, and it is therefore possible to create objects of these two subclasses.

Please note that if a subclass does not define all `abstract` methods defined in the `abstract` superclass, the subclass is still `abstract` because it still has `abstract` methods, and it has to be marked `abstract` or compile-time errors will occur.

The elegance of defining `abstract` methods is that if the `getWidth()` and `getHeight()` methods of the `Shape` class are not defined `abstract`, it is necessary to provide method bodies for these two methods, such as:

```
public int getWidth() {
    return -1;
}

public int getHeight() {
    return -1;
}
```

The values of the width and height of a shape must be positive, and the return value `-1` indicates there is a problem with the object. As the two methods are not `abstract` methods, the developer of the subclass has no obligation to define these two methods. Then, calling these two methods of a subclass object of the parent class `Shape` may return `-1` at runtime, which is unreasonable, and the software application may give wrong results.

Therefore, defining `abstract` methods forces the developer of the subclass to determine a proper way to perform a particular behaviour, or

the subclass must be kept abstract and no object of this subclass can be created.

Abstract class with concrete subclasses

As mentioned above, abstract classes are used to define a general type, such as the `Shape` class mentioned in the reading, with behaviours that are undefined. It is therefore impossible to create an object of an abstract class. For a specific type of the general type, such as `Rectangle`, all its behaviours are known and can be defined properly, and it is then possible to create an object of a specific type.

We said that a specific type is also a general type, such as a `Rectangle` is a `Shape`; it is therefore a natural consequence that a specific type is defined to be a subclass of the general type. Furthermore, as the general type is defined with some undefined behaviours, which are abstract methods, it is an abstract class. By contrast, the class definition of the specific type must define all abstract methods that are inherited from the abstract superclass, so that it is possible to create an object of the specific type. Therefore, a specific type is usually known as concrete subclass of the abstract superclass.

At runtime, a program that is written to manipulate a general type by sending messages to the object referred via a variable of an abstract class type actually sends the messages to an object of a concrete subclass of the abstract superclass. The reason is that we said that a variable of type class `T` can refer to an object of class `T` and any subclass of class `T`. In our case, the class `T` is abstract and is therefore no object of the type class `T`. As a result, an object referred by a variable of type class `T` must be an object of a concrete subclass of the class `T`.

In the following sections, we discuss the applications of the abstract superclass with concrete classes. You will then realize the necessity of such implementation.

Modelling a pet shop using abstract classes

A simple and interesting example that needs an `abstract` class is the problem of modelling a pet shop. A pet shop has a collection of pets, and each can make a sound or speak in its own way. You don’t know how a pet speaks if you are not sure whether it is a dog or a cat, but you are sure that a pet can make sounds in its own way. Furthermore, you feel sure that the way a pet says a few words is that it says a word a few times. Therefore, we can figure out that each pet has two behaviours: `sayAWord` and `sayAFewWords`. The behaviour `sayAFewWords` is well defined, but the behaviour `sayAWord` is not.

Based on the above analysis, a possible way to define the class `Pet` is shown in Figure 7.76.

```
// Definition of class Pet
public abstract class Pet {
    // No Attributes

    // Abstract method to be defined in concrete subclasses
    public abstract void sayAWord();

    // The concrete way a pet to say a few word.
    public void sayAFewWords() {
        // Say arbitrary number of words
        int count = (int) (Math.random() * 10) + 1;
        // Use a for loop to say a word for a few times
        for (int i=0;i < count; i++) {
            sayAWord();
        }
        // Skip to next new lines
        System.out.println();
    }
}
```

Figure 7.76 Pet.java

The `Pet` class models a group of general pet objects. As the actual behaviour of how a pet says a word is undefined until the pet is specified, the method `sayAWord()` is marked `abstract`. But, no matter how a pet says a word, the detailed operations of saying a few words by a pet can be defined based on the conceptual or abstract behaviour `sayAWord` as in the `sayAFewWords()` method in the `Pet` class.

As mentioned, an `abstract` class defines a conceptual and non-existent object. Therefore, it is not possible to create an object of `Pet` class. The following statement would cause a compile-time error.

```
new Pet()
```

The `abstract Pet` class is written as a general class of some specific classes, such as `Cat` and `Dog`. These specific classes are defined as the subclasses of the `abstract` superclass, and their class definitions must define the `abstract sayAWord()` method so that an object can be created.

Cats and dogs are pets, and it is obvious that ‘a cat is a pet’ and ‘a dog is a pet’. It is therefore reasonable to define the classes `Cat` and `Dog` as subclasses of the superclass `Pet`. We know that a pet can say a word and say a few words. The `abstract Pet` class already defines the common behaviour `sayAFewWords`; what remains to be defined is how the two specific types define their own way of the behaviour `sayAWord`.

The classes `Cat` and `Dog` can be defined as shown in Figure 7.77 and Figure 7.78.

```
// Definition of class Cat
public class Cat extends Pet {
    // No Attributes

    // Method

    // Define the abstract behaviour for a cat to say a word
    public void sayAWord() {
        System.out.print("Meow! ");
    }
}
```

Figure 7.77 Cat.java

```
// Definition of class Dog
public class Dog extends Pet {
    // No Attributes

    // Method

    // Define the abstract behaviour for a Dog to say a word
    public void sayAWord() {
        System.out.print("Bark! ");
    }
}
```

Figure 7.78 Dog.java

Both classes `Cat` and `Dog` inherit the methods from the superclass `Pet`, including the `abstract sayAWord()` method. The subclasses provide concrete definitions of the `abstract` method and are therefore non-`abstract`. The inheritance relationships among the involved classes are shown in Figure 7.79.

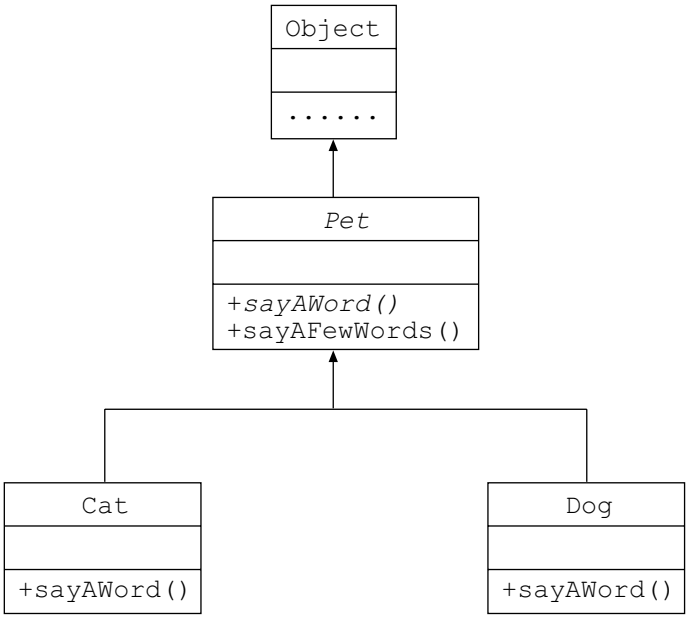


Figure 7.79 The inheritance relationships among the `Pet, Cat, Dog` and the implicit ultimate superclass `Object`

(In Figure 7.79, the class name `Pet` and the method name `sayAWord()` are set in italics to indicate that they are `abstract`.)

Based on the definitions of classes `Cat` and `Dog`, it is then possible to create objects of classes `Cat` and `Dog`, such as:

```
Cat cat = new Cat();
Dog dog = new Dog();
```

The classes `Cat` and `Dog` define their own `sayAWord()` methods and inherit the common `sayAFewWords()` method from the superclass `Pet`. Therefore, both a `Cat` object and a `Dog` object can accept messages `sayAWord` and `sayAFewWords` to perform the corresponding behaviours. That is, it is possible to have a program segment such as:

```
cat.sayAWord();
cat.sayAFewWords();
dog.sayAWord();
dog.sayAFewWords();
```

The following program segment is less trivial than the above but is perfectly valid.

```
Pet pet = new Cat();
pet.sayAWord();
pet.sayAFewWords();
```

Even though the `Pet` class is abstract, it can be a variable type to refer to objects that have members defined or declared by the `Pet` class; that is, objects of the concrete subclasses of the `Pet` class. A `Cat` object is created and is referred by the variable of type `Pet`. The statement is acceptable because a `Cat` object can be treated as a `Pet` object, even though it is clear that there is no real `Pet` class object. Furthermore, with a `Pet` variable, it is possible to send the messages `sayAWord` and `sayAFewWords` to the referred object, because the `Pet` class defines these methods. Briefly, an abstract class variable can refer to the concrete subclass objects.

When the `Cat` object receives the message `sayAWord` via the variable `pet`, the method to be executed is the `sayAWord()` method defined in the `Cat` class, and the following message is shown on the screen.

```
Meow!
```

Afterwards, the message `sayAFewWords` is sent to the `Cat` object via the variable `pet`. The method to be executed is the `sayAFewWords()` method inherited from `abstract` superclass `Pet`.

```
// The concrete way a pet to say a few word.
public void sayAFewWords() {
   // Say arbitrary number of words
   int count = (int) (Math.random() * 10) + 1;
   // Use a for loop to say a word for a few times
   for (int i=0;i < count; i++) {
      sayAWord();
```

```
    }
    // Skip to next new lines
    System.out.println();
  }
```

While the `Cat` object is executing the `sayAFewWords()` method, the statement in the `for` loop sends `sayAWord` messages to itself (the statement is implicitly `this.sayAWord()`), so that it executes its `sayAWord()` method. For a `Cat` object, the `sayAWord()` method to be executed is the one defined in the `Cat` class, and it is executed a few times in the `for` loop (according to the random number obtained). If the random number obtained by the expression

```
(int) (Math.random() * 10) + 1
```

is five, the `sayAFewWords()` method displays the following output on the screen.

```
Meow! Meow! Meow! Meow! Meow!
```

The sequence of method calls involved can be visualized in Figure 7.80.

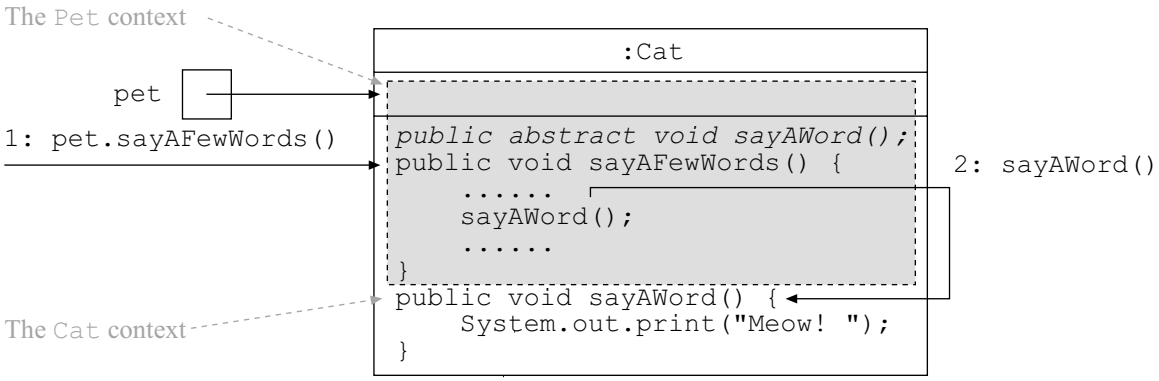


Figure 7.80 The sequence of method calls for statement pet.sayAFewWords()

Figure 7.80 illustrates that the `sayAFewWords()` method defined in the `Pet` class calls the concrete `sayAWord()` method defined in the concrete subclass. You should notice that if the `Pet` class did not define the abstract `sayAWord()` method, the `Pet` class would have caused compile-time errors. While the compiler software compiles the `Pet` class, it finds that the `sayAFewWords()` method needs to call another `sayAWord()` method, and the abstract method `sayAWord()`'assures' the compiler software that a concrete `sayAWord()` will be provided in a concrete subclasses and the objects of a concrete subclass must have a concrete `sayAWord()` method.

With the definitions of classes `Pet`, `Cat` and `Dog`, it is now possible to define a `PetShop` class as shown in Figure 7.81 that manipulates a list of `Pet` objects.

```
// Definition of class PetShop
public class PetShop {

    // Main executive method
    public static void main(String args[]) {
        // Create a collection of two pets, a cat and a dog
        Pet[] pets = {
            new Cat(),
            new Dog()
        };

        // Request each pet to say a few words
        for (int i=0; i < pets.length; i++) {
            pets[i].sayAFewWords();
        }
    }
}
```

Figure 7.81 PetShop.java

Although it is not possible to create an object of the `abstract` class `Pet`, we can have variables of type `Pet`. The consequence is that we can define an array with the element type `Pet`. In the `main()` method of the `PetShop` class, the statement

```
Pet[] pets = {
    new Cat(),
    new Dog()
};
```

declares a variable `pets` of the type array with the element type `Pet` and is initialized with an array object that refers to two `Pet` objects, a `Cat` object and a `Dog` object. It is equivalent to the following statements:

```
Pet[] pets = new Pet[2];
pets[0] = new Cat();
pets[1] = new Dog();
```

The scenario is visualized as shown in Figure 7.82.

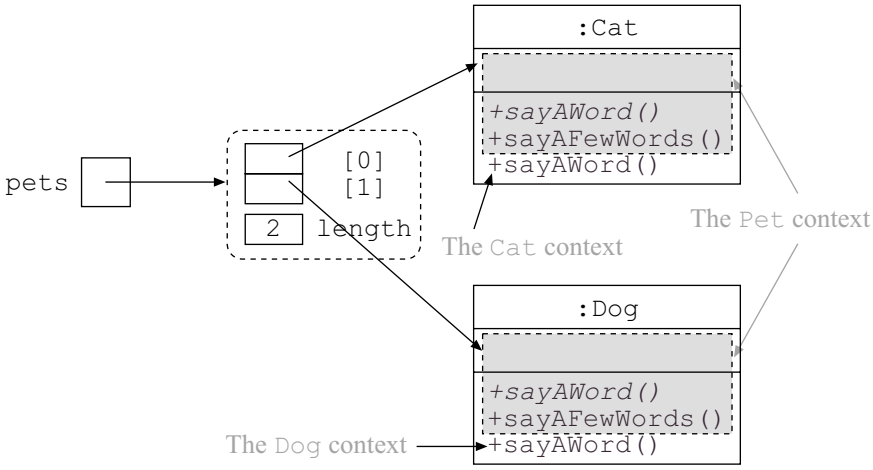


Figure 7.82 The `Cat` object and the `Dog` object referred to by the array object with element type `Pet` (The contexts for `Object` class are not shown, for simplicity.)

The statement in the `for` loop in the `main()` method repeatedly sends the `sayAFewWords` messages to the objects referred by the array object. When the `Cat` object and the `Dog` object receive the message, they execute the `sayAFewWord()` method defined in the `Pet` class and subsequently call their own `sayAWord()` method. Therefore, by compiling the classes and executing the `PetShop` class, one possible output to be shown on the screen is:

```
Meow! Meow! Meow! Meow! Meow! Meow! Meow! Meow! Meow!
Bark! Bark! Bark! Bark! Bark!
```

The number of words shown on the screen depends on the random numbers obtained in the `sayAFewWords()` method, and each execution of the `PetShop` class may show a different output.

Polymorphism is used in the above example, and the `for` loop in the `main()` method manipulates the objects referred by the array object as if they are general `Pet` objects and they can behave differently. This illustrates the possibility of writing a class that uses an abstract class in the class definition, but it is actually manipulating objects of concrete subclasses of the abstract superclass at runtime.

Calculation of MPF using abstract classes

The above example illustrates the use of `abstract` class, but it seems unrealistic to write programs just for modelling real-world pets that are making sounds. To help you understand the application of abstract classes, the following example concerns the problem of the calculation of payroll with the Mandatory Provident Fund (MPF).

The requirement of the problem is that at the end of the month, the human resources department of a company has to determine the total payroll and MPF contributions made by the company and the staff. If the salary of a staff member is less than HK$5,000, the company contributes 5% of the staff salary as the MPF, and the staff member can retain the entire salary. Otherwise, both the company and the staff member have to contribute an amount of 5% of the staff salary, and the staff member can only get 95% of the original salary. The maximum contributions by the staff member and the company are HK$1,000 each. The staff members in the company are categorized into clerk, salesperson and manager. The calculations of different categories are provided as follows:

1 The salaries of clerks are a fixed basic salary.

2 The salary of a salesperson is the sum of a fixed basic salary and the sales volume at a fixed commission rate.

3 The salary of a manager is the sum of a fixed basic salary and a fixed cash allowance.

No matter which payroll calculation method is used, the rules governing the calculation of MPF are the same. The way to determine the salary of a staff member depends on the staff type. To solve the problem, we can

define an abstract `Staff` class with the concrete subclasses `Clerk`, `SalesPerson` and `Manager`. The inheritance relationships among them can be visualized in Figure 7.83.

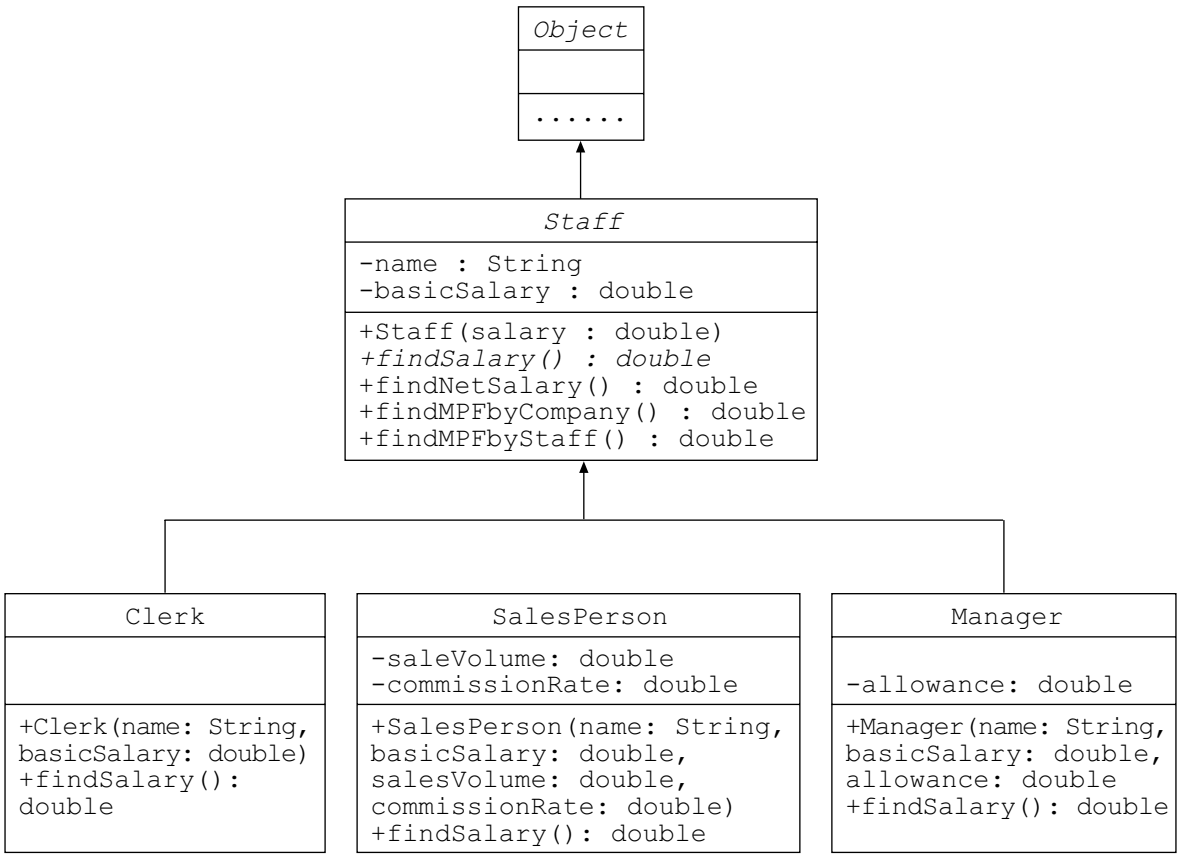


Figure 7.83 The inheritance relationships among Staff, Clerk, SalesPerson and Manager

Although it is not possible to create objects of an `abstract` class, an `abstract` class can define its own constructor and will be called by the `super()` statement (either implicitly or explicitly) by a concrete subclass constructor. Based on the class designs shown in Figure 7.83, the classes `Staff`, `Clerk`, `SalesPerson` and `Manager` are defined as shown in Figures 7.84 to 7.87 respectively.

```
// Definition of abstract class Staff
public abstract class Staff {
   // Attributes
   private String name;
   private double basicSalary;

   // Constructor
   public Staff(String name, double basicSalary){
      this.name = name;
      this.basicSalary = basicSalary;
   }

   // Methods
```

```
    // Find the raw salary
    public abstract double findSalary();

    // Get the staff name
    public String getName() {
         return name;
    }

    // Get the basic salary
    public double getBasicSalary() {
       return basicSalary;
    }

    // Find the net salary after MPF contribution
    public double findNetSalary() {
       return (findSalary() - findMPFByStaff());
    }

    // Find the MPF contribution by the company
    public double findMPFByCompany() {
       return (findSalary() > 20000.0) ? 1000.0 : findSalary() * 0.05;
    }

    // Find the MPF contribution by the staff
    public double findMPFByStaff() {
       double salary = findSalary();
       if (salary < 5000.0) {
          return 0.0;
       }
       else if (salary >= 20000.0) {
          return 1000.0;
       }
       else {
          return salary * 0.05;
       }
    }
}
```

Figure 7.84 Staff.java

```
// Definition of class Clerk
public class Clerk extends Staff {
   // No Attribute

   // Constructor
   public Clerk(String name, double basicSalary){
      super(name, basicSalary);
   }

   // Method

   // Find the raw salary
   public double findSalary() {
      return getBasicSalary();
   }
}
```

Figure 7.85 Clerk.java

```
// Definition of class SalesPerson
public class SalesPerson extends Staff {
   // Attributes
   private double salesVolume;
   private double commissionRate;

   // Constructor
   public SalesPerson(String name, double basicSalary,
                      double salesVolume, double commissionRate) {
     super(name, basicSalary);
     this.salesVolume = salesVolume;
     this.commissionRate = commissionRate;
   }

   // Find the raw salary
   public double findSalary() {
     return getBasicSalary() + salesVolume * commissionRate;
   }
}
```

Figure 7.86 SalesPerson.java

```
// Definition of class Manager
public class Manager extends Staff {
    // Attributes
    private double allowance;

    // Constructor
    public Manager(String name, double basicSalary, double allowance){
        super(name, basicSalary);
        this.allowance = allowance;
    }

    // Find the raw salary
    public double findSalary() {
        return getBasicSalary() + allowance;
    }
}
```

Figure 7.87 Manager.java

The rules governing the calculations of the MPF contributions and the net salary of a staff member apply to all staff. Therefore, the methods `findNetSalary()`, `findMPFByStaff()` and `findMPFByCompany()` are defined in the common `abstract` superclass `Staff`, and the `findSalary()` method is left to be defined in the concrete subclasses.

With the above classes, a sample calculation of the payroll and the MPF contributions in a company can be done with the class `PayrollCalculator1` as shown in Figure 7.88.

```
// Definition of class PayrollCalculator1
public class PayrollCalculator1 {
    // Attribute
    private Staff[] staffList;    // The staff list to be processed

    // Constructor
    public PayrollCalculator1(Staff[] staffList) {
        this.staffList = staffList;
    }

    public void showReport() {
        // Declare and initialize running totals
        double totalSalary = 0.0;
        double totalMPFByStaff = 0.0;
        double totalMPFByCompany = 0.0;

        // Show the listing title
        System.out.println("Staff\tRaw\tNet\tMPF by\tMPF by");
        System.out.println("Name\tSalary\tSalary\tStaff\tCompany");
        System.out.println("------- ------ ------ ------ -------");

        // Iterate each staff to show his/her details
        for (int i=0; i < staffList.length; i++) {
            // Get the staff payroll details
            double salary = staffList[i].findSalary();
            double netSalary = staffList[i].findNetSalary();
            double mpfByStaff = staffList[i].findMPFByStaff();
            double mpfByCompany = staffList[i].findMPFByCompany();

            // Show the details
            System.out.println(
                staffList[i].getName() +
                "\t" + salary +
                "\t" + netSalary +
                "\t" + mpfByStaff +
                "\t" + mpfByCompany);

            // Update the running totals
            totalSalary += netSalary;
            totalMPFByStaff += mpfByStaff;
            totalMPFByCompany += mpfByCompany;
        }

        // Show the grand totals
        System.out.println("------ ------ ------ ------- -------");
        System.out.println(
            "Total" +
            "\t\t" + totalSalary +
            "\t" + totalMPFByStaff +
            "\t" + totalMPFByCompany);
    }
}
```

Figure 7.88 PayrollCalculator1.java

To test the `PayrollCalculator1` class, a driver program `TestPayrollCalculator1` is written as shown in Figure 7.89.

```
// Definition of class TestPayrollCalculator1 {
public class TestPayrollCalculator1 {

    // Main exeuctive method
    public static void main(String args[]) {
      // Create an array of staff in the company
      Staff[] staff = {
         new Clerk("Mary", 4000.0),
         new Clerk("Peter", 6000.0),
         new SalesPerson("Joe", 4000.0, 100000.0, 0.05),
         new SalesPerson("Amy", 5000.0, 200000.0, 0.08),
         new Manager("John", 20000.0, 10000.0)
      };

      // Create a PayrollCalculator1 object by supplying an array
      // of Staff objects
      PayrollCalculator1 calculator = new PayrollCalculator1(staff);

      // Show the payroll report
      calculator.showReport();
   }
}
```

Figure 7.89 TestPayrollCalculator1.java

The `main()` method of the `TestPayrollCalculator1` class creates an array variable `staff` with array elements of the type `Staff` and initializes it with an array object referring to five objects of concrete subclasses of the abstract superclass `Staff`.

```
Staff[] staff = {
  new Clerk("Mary", 4000.0),
  new Clerk("Peter", 6000.0),
  new SalesPerson("Joe", 4000.0, 100000.0, 0.05),
  new SalesPerson("Amy", 5000.0, 200000.0, 0.08),
  new Manager("John", 20000.0, 10000.0)
};
```

As the constructors of the concrete subclasses differ in the parameter lists, different lists of values are supplied to the constructors to set up the objects accordingly. For example, the clerk category staff member Mary has a fixed basic salary of HK$4,000. The salesperson Amy has a fixed basic salary HK$5,000 and her sales volume in the month is HK$200,000 at a fixed commission rate 8%. For the time being, the staff list is written as part of the `main()` method. When you learn how to perform file input/output in *Unit 8*, you can write a similar program that reads a staff list from a file stored in your hard disk.

The `main()` method of the `TestPayrollCalculator1` then creates a `PayrollCalculator1` object by supplying the reference of

the array object that is referring to five `Staff` objects (more exactly, five objects of concrete subclasses of the abstract superclass `Staff`).

```
PayrollCalculator1 calculator = new PayrollCalculator1(staff);
```

Finally, the `PayrollCalculator1` object is required to show the payroll report:

```
calculator.showReport();
```

During the execution of the `for` loop in the `showReport()` method, the following statements obtain the necessary information about an object of a `Staff` subclass:

```
double salary = staffList[i].findSalary();
double netSalary = staffList[i].findNetSalary();
double mpfByStaff = staffList[i].findMPFByStaff();
double mpfByCompany = staffList[i].findMPFByCompany();
```

The `findSalary()` method is declared `abstract` in the `Staff` class, and the method is provided in a concrete subclass of the `abstract` superclass `Staff`. For the remaining three methods, the calculations are the same for all types of staff, and their implementations obtain the salary of a staff member by calling the `findSalary()` method defined in a concrete subclass.

Compiling the above classes and executing the class `TestPayrollCalculator1` shows the following output on the screen.

```
Staff   Raw     Net     MPF by  MPF by
Name    Salary  Salary  Staff   Company
------- ------- ------- ------- -------
Mary    4000.0  4000.0  0.0     200.0
Peter   6000.0  5700.0  300.0   300.0
Joe     9000.0  8550.0  450.0   450.0
Amy     21000.0 20000.0 1000.0  1000.0
John    30000.0 29000.0 1000.0  1000.0
------- ------- ------- ------- -------
Total           67250.0 2750.0  2950.0
```

With polymorphism, the `PayrollCalculator1` class `showReport()` method manipulates all staff members as general `Staff` objects. It is also possible to obtain the same result without using abstract classes and polymorphism. Please refer to Appendix I for the implementation of calculating MPF without abstract classes and polymorphism. Then you will appreciate the beauty of polymorphism and the use of abstract classes.

Self-test 7.10

Define a `TempStaff` class as a subclass of the abstract superclass `Staff`. There is no base salary for a temporary staff member, and his or her salary is the hourly wage times the number of working hours in the month. Modify the `main()` method of the `TestPayrollCalculator1` class so that the staff list includes a temporary staff member Benny, whose number of working hours in the month is 160 and hourly wage is HK$50.

Deriving `abstract` classes from existing classes

You should now understand the use of abstract classes and their applications. You may need some time to become familiar with it. In the design phase of the software development cycle, an experienced programmer can figure out whether abstract classes are needed or are beneficial to the software design, and concrete subclasses can be derived subsequently. This is kind of a 'top-down' approach. Beginners can investigate the existing classes to determine whether there is a necessity of an abstract class from a 'bottom-up' approach. Later on, when you become familiar with the application of abstract classes, the idea of using abstract classes will probably come to mind.

For example, while you are analysing a banking problem, it is found that the classes `SavingsAccount` and `CurrentAccount` are required to model real-world saving accounts and current accounts. Furthermore, you determine the class designs are as shown in Figure 7.90.

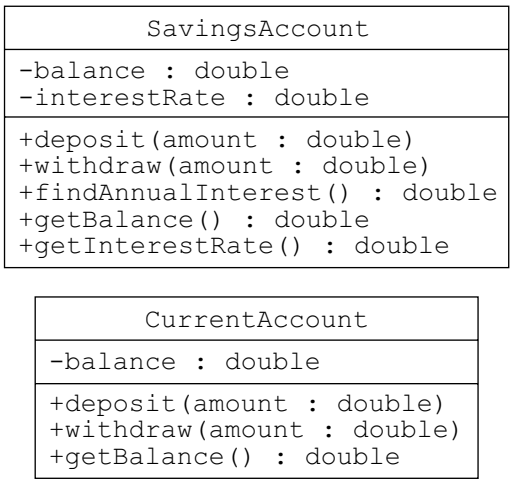


Figure 7.90 The designs of classes SavingsAccount and CurrentAccount

It is found that the attribute `balance` and three methods of the two class designs are common. The question to investigate is whether a `SavingsAccount` object can be considered to be a `CurrentAccount`. That is, you should determine whether there is an 'is a' relationship between them. The answer to the question is 'no',

because a `SavingsAccount` is not a `CurrentAccount` and vice versa. They are two independent account types.

Then, you can further ask yourself whether they are specific types of a general type. The answer is 'yes', and both `SavingsAccount` and `CurrentAccount` can be considered specific types of a general type `Account`. Therefore, the class `SavingsAccount` and `CurrentAccount` can be defined as subclasses of superclass `Account`.

The next step is to consider the methods one by one. If the interpretations of the same method for both subclasses are the same, it makes sense to extract the method from the subclasses and define it in the common superclass. If the subclasses have a common behaviour but it behaves differently, the method can be declared as an `abstract` method in the abstract superclass, and the concrete subclasses define their own concrete methods.

With respect to the `SavingAccount` and `CurrentAccount`, the business rules for performing the behaviours `deposit` and `withdraw` are different for the two account types. Therefore, the methods `deposit()` and `withdraw()` are declared `abstract` in the superclass `Account`. By contrast, the operation of the `getBalance()` method is the same for both classes and is therefore declared in the superclass `Account`. As annual interest is only applicable to a savings account, the attribute `interestRate` and its getter/setter methods are declared in the specific type, the `SavingAccount` class.

Based on the above investigation, the classes `SavingsAccount` and `CurrentAccount` can be modified, and a new abstract superclass `Account` is introduced, as in Figure 7.91.

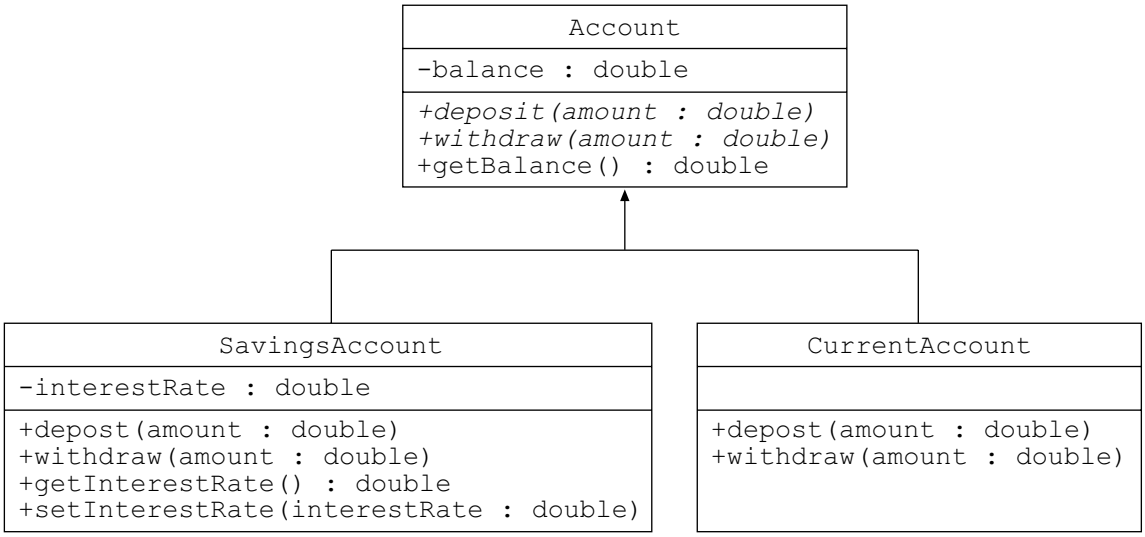


Figure 7.91 A possible design of the classes Account, SavingsAccount and CurrentAccount

Figure 7.91 shows a possible design according to the business operations. The way to determine the `abstract` superclass depends on your

understanding of the problem. The more you understand the problem, the better design you can have.

Self-test 7.11

A developer is developing a software application for bookkeeping of the telephone calls received by the staff in the customer service department. The calls can be a general enquiry about services, a service request or a complaint concerning service.

1 Determine the attributes and behaviours of the three different telephone calls.

2 Investigate their similarities, and determine the `abstract` superclass and concrete subclasses that can be used to model them.

Defining abstract classes for future subclass development

An abstract class can have concrete methods that define common behaviours of the general type. Some behaviours are declared `abstract` and are left to be defined by the concrete subclasses. Therefore, abstract classes can be used a blueprint for the subclasses, and the concrete subclasses are mutually independent. As a result, it is possible to define abstract classes for future subclass development.

Currently, some software development models with the Java programming language are designed by experts and are implemented as abstract classes and its variant, Java `interface` (discussed in *Unit 9*). Those experts also provide sample concrete subclasses as proof of concepts, and other software developers write their own concrete subclasses for their own software projects. Such a software development model facilitates an excellent distribution of work, because the experts design the common development specifications and the other software developers can concentrate their work on their own specific projects.

For example, some software experts designed the Java servlet specification to be used for dynamic Web page generation with Java technologies. They defined the way to develop Java servlets as abstract classes that can be handled by Java Web server software. Then, you do not have to develop the Web server by yourselves but just write your own concrete classes of the abstract classes; the Java Web server can generate the Web pages according to your concrete classes. Figure 7.92 shows the relationships.

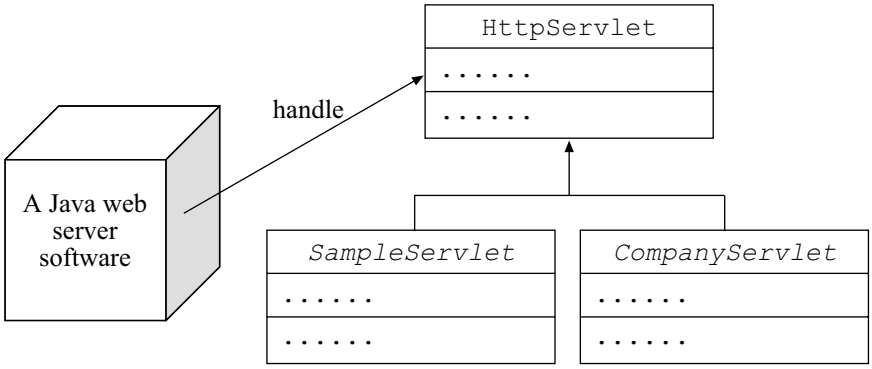


Figure 7.92 Java web server software that can handle a general (abstract) `HttpServlet`, which can handle the concrete subclasses `SampleServlet` and `CompanyServlet`

In Figure 7.92, the cube represents Java Web server software developed by a software company with the Java programming language that is thoroughly tested. It is built to manipulate a general `HttpServlet`, and a sample concrete subclass `SampleServlet` is provided to prove that the software works properly. Then, your in-house software development team can purchase the software and just develop your own concrete subclasses, such as the `CompanyServlet` (as shown in Figure 7.92) for your official company websites. The company does not have to be concerned with how to develop the Web server software or how it is developed.

Furthermore, other software companies can develop their own Web server software that can handle the abstract `HttpServlet` class objects. If you find another Web server software is better (such as more reliable or simply cheaper) than the one you are currently using, you can simply switch the software; you do not have to modify a single statement in the `CompanyServlet` class. The reason is that if the software can handle the abstract `HttpServlet` class, it must be able to handle your concrete subclass `CompanyServlet` properly.

In *Unit 8*, we will discuss how to perform input/output operations with the Java programming language. Then, we will discuss some abstract classes that are defined in the software library that come with the Java Runtime Environment (JRE). These abstract classes define the common behaviours of an object that read data from a data source or write data to a data destination. Some concrete subclasses are defined in the software library that can perform reading data from, and writing data to, a specific media or channel. A file stored in a hard disk is a common data source or data destination. Furthermore, it leaves room for software developers to write specific subclasses of the abstract superclass to perform reading/writing operations on new media or channels.

Summary

This has been a rather lengthy unit that contains plenty of sample class definitions. Each sample class definition demonstrates a single feature. You should pay attention to the differences among them, so that you can understand the effects of the changes.

We first discussed two important programming concepts — information hiding and encapsulation. They are implemented by marking attributes `private` and providing corresponding `public` getter/setter methods. The idea of information hiding suggests that object attributes should be kept out of reach from the users of the object, so that it is not possible to assign invalid values to the attributes. The concept of encapsulation recommends that internal operations are transparent to the users of the class, so that it is not necessary for the users of the class to know its internal operations. Then, if its internal operations need modifications, the changes are localized and no other classes need modifications.

There are a few steps in initializing an object during its creation process. In previous units, we mentioned there were implicit and explicit initializations of the object attributes. In this unit, we further discussed how a special method — the constructor — is called during the creation process. Its format is similar to a normal method except the method name is identical to the class name and it defines no return type. Moreover, we mentioned that it is possible to use a `this()` statement with a suitable parameter value list in a constructor to call another constructor in the same class. Besides calling another constructor in the class, the keyword `this` can be considered an imaginary attribute of the object that is referring to the object itself. It is commonly used for resolving attributes or methods of the object, and it can be supplied to a method call as the reference of the object.

The Java programming language features inheritance, which enables a class to be derived based on an existing class. In other words, the definition of the derived class (a subclass) extends the definition of the existing class (the superclass) with an `extends` clause in the class definition. Such a feature enables a subclass to reuse the definition of the superclass and hence minimizes duplicated code by the ‘copy-and-paste’ approach. Furthermore, there is an inheritance relationship between the subclass and the superclass in which the subclass inherits all attributes and methods from the superclass. The conceptual relationship behind inheritance is the ‘is a’ relationship. The inheritance relationship between the subclass and the superclass discloses that a subclass ‘is a’ superclass. As a result, a subclass object can be considered a superclass object. At the same time, we mentioned the ‘has a’ relationship indicates the possession relationship between two classes.

It is possible to have two methods of the same name. If a class definition has two different methods with the same name but different parameter lists, it is considered to be overloading. The one to be executed is determined by the parameter value list supplied to the method. If a subclass defines a method with the same name and parameter list as a

method in the superclass, it is considered to be overriding, and the method defined in the subclass overrides the one defined in the superclass. If a subclass object receives a message and the message corresponds to a method defined in the superclass and the subclass does not define such a method, the method defined in the superclass is executed. Otherwise, the method defined by the subclass overrides a method defined in the superclass, and the subclass overriding method will be executed instead.

Polymorphism is the ability for objects that are subclasses of the same parent class to behave differently on receiving the same message that is defined by the common parent class. In the Java programming language, it is implemented as a variable of a superclass type, say class `T`, that can refer to an object of any subclass of class `T`. Then, if a message is sent to the object via the variable of type class `T`, the behaviour (or the method to be executed) is determined by the actual class of the object (that is, a subclass of class `T` or class `T` itself), not by the type of the variable.

Even though a constructor is like a method, they are handled differently with respect to inheritance. A subclass does not inherit constructors from its immediate superclass, and a `super()` statement is required in the subclass constructor to call the superclass constructor to initialize the superclass context of the object. An implicit `super()` statement is added if there is no `this()` statement in the constructor. If the superclass provides no constructor with an empty parameter list, the subclass must use a `super()` statement with a suitable parameter list to call a superclass constructor explicitly. Furthermore, the keyword `super` can be used to refer to the overridden members of the superclass.

The three key features of an object-oriented programming language — encapsulation, inheritance and polymorphism — are discussed in the unit with their contributions in promoting writing better software and improving software productivity and maintainability. It is much more important for you to realize their significance and the ways to use them in object-oriented programming.

Appendix A: `final` methods and `final` classes

Polymorphism is achievable because methods in the superclass can be overridden in the subclasses and a method to be executed by an object is based on its real or actual class. Occasionally, it is undesirable for a method to be overridden in the subclasses. Then, you can mark the method `final`. For example, if the `print()` method of the `Date1` class is defined as

```
public final void print() {
   System.out.print(year);
   System.out.print((month < 10) ? "0" + month : "" + month);
   System.out.print((day < 10) ? "0" + day : "" + day);
}
```

it is not possible to define a subclass with a `print()` method with an empty parameter list, or a compilation error will occur.

A benefit of marking a method `final` is for optimization purposes, because a `final` method cannot be overridden in the subclass, and hence the compiler software can determine that the method to be called must be the one defined in the class. Then, it is not necessary to determine the method to be called at runtime, and the overhead of the determination can be avoided, thereby leading to faster execution.

If all methods in a class have to be marked `final`, another way is to mark the class `final`. It is then not possible to derive a subclass based on a `final` class, and all methods in a `final` class are by default `final`. For example, if the `Date1` class is marked `final`, such as,

```
public final class Date1 {
    ......
}
```

it is a compilation error to derive a subclass, say `BritishDate1`, based on the `Date1` class.

```
public class BritishDate1 extends Date1 {
    ......
}
```

Then, if you have a variable `date` of type `Date1` and it stores a reference to an object, the object must be a `Date1` object because there is no subclass of a `final` class. Furthermore, the methods executed by an object that is referred by a `Date1` variable must be those defined in the `Date1` class.

Appendix B: the `Object` class

You learned that the syntax of defining a class is:

```
[public] [final] class Class-name [extends Superclass-name] {
    ......
}
```

The `extends` clause is optional. However, if the class is defined without the `extends` clause, it is implicitly a subclass of the class `java.lang.Object`. That is, if a class definition has no `extends` clause, such as,

```
public class NewClass {
    ......
}
```

it is equivalent to the following one:

```
public class NewClass extends java.lang.Object {
    ......
}
```

As the compiler software implicitly resolves classes defined in the `java.lang` package, the `java.lang.Object` class is understood to be simply `Object` in the source code. Therefore, the above class definition can be alternatively written as:

```
public class NewClass extends Object {
    ......
}
```

Therefore, except for the `Object` class itself, all classes in the Java programming language are the subclasses of the `Object` class, either directly (class definitions without the `extends` clause or explicitly extending the `Object` class) or indirectly (subclasses of the other classes that are subclasses of the `Object` class directly or indirectly).

Please use the following reading to gain more understanding of the `Object` class.

Reading

King, section 11.5, pp. 457–61

From the reading, you know that all classes are subclasses of the `Object` class, and they possess all methods defined in the `Object` class. Among the methods defined in the `Object` class, two are commonly used and are usually overridden in the subclasses. They are the `equals()` method and the `toString()` method.

The `equals()` method tests whether two objects are equal or equivalent. As it is a method defined in the `Object` class, all classes have such a method. For example, the class `TestEquals1` written as shown in Figure 7.93 examines the `equals()` method of the `TicketCounter4` class.

```
// Definition of class TestEquals1
public class TestEquals1 {

   // Main executive method
   public static void main(String args[]) {
      // Create two TicketCounter4 objects and are referred
      // by variable counter1 and counter2
      TicketCounter4 counter1 = new TicketCounter4(10);
      TicketCounter4 counter2 = new TicketCounter4(10);

      // Show the results of == and equals() method
      System.out.println(counter1 == counter2);
      System.out.println(counter1.equals (counter2));
      System.out.println(counter2.equals (counter1));
   }
}
```

Figure 7.93 TestEquals1.java

The `main()` method of the `TestEquals1` class first creates two `TicketCounter4` objects as visualized in Figure 7.94.

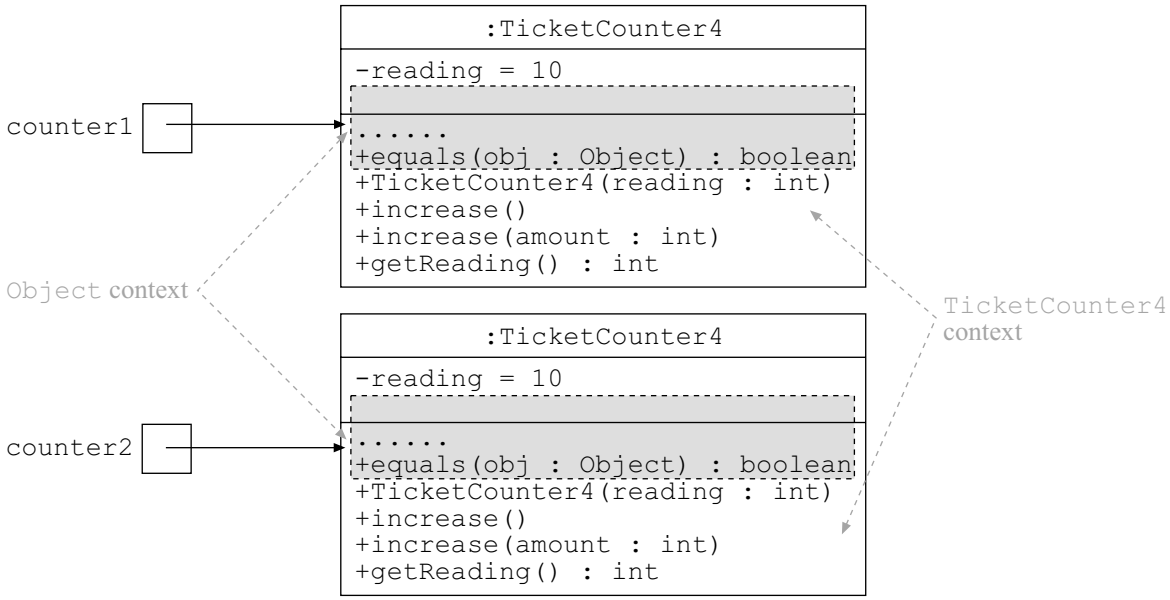


Figure 7.94 The two created `TicketCounter4` objects referred to by variables `counter1` and `counter2`

The subsequent statements in the `main()` method show the results of the following three expressions:

```
counter1 == counter2
counter1.equals(counter2)
counter2.equals(counter1)
```

For the first expression, the `==` operator verifies whether the contents of the two variables `counter1` and `counter2` are the same. As the contents of non-primitive variables are object references, the contents are equal if they are referring to the same object. As the variables `counter1` and `counter2` are referring to two different `TicketCounter4` objects, their contents must be different, and the expression is therefore `false`.

The second expression is the return value of the `equals()` method executed by the `TicketCounter4` object referred to by the variable `counter1`, and the content of the variable `counter2` is supplied to the `equals()` method. The `TicketCounter4` class defines no `equals()` method, and the `equals()` method to be executed is the one inherited from its superclass, which is the implicit superclass `Object`. As mentioned in the reading, the inherited `equals()` method from the `Object` class simply verifies whether the two variables `counter1` and `counter2` are referring to the same object, which is practically equivalent to the `==` operator. Therefore, the return value of the `equals()` method is `false` as well.

The third expression is similar to the second one. A difference is that the message `equals` is sent to the `TicketCounter4` object referred to by the variable `counter2` instead of the `counter1`. Therefore, it is the `TicketCounter4` referred by the variable `counter2` that performs the `equals` behaviour instead of the one referred by the variable `counter1`. It makes sense that the results `counter1.equals(counter2)` and `counter2.equals(counter1)` are always the same. As the object referred by the `counter2` variable is also a `TicketCounter4` object, and if the `TicketCounter4` class does not define an `equals()` method, the `equals()` method to be executed is the one inherited from the `Object` class. The return value of the `equals()` method is `false` because variables `counter1` and `counter2` are referring to two different `TicketCounter4` objects. As a result, compiling the classes and executing the `TestEquals1` class, it shows the output:

```
false
false
false
```

To define an `equals()` method for a class, you should first ask yourself what criteria are being considered to be equal or equivalent to the two objects. Two `TicketCounter4` objects are considered to be equal if the values of their attribute `reading` are the same. For example, the class `TicketCounter6` as shown in Figure 7.95 is defined based on the `TicketCounter4` with an overriding `equals()` method.

```
// Definition of class TicketCounter6
public class TicketCounter6 {
   // Attribute
   public static final int LIMIT = 1000;
   private int reading;  // The reading of the counter

   // The constructors
   public TicketCounter6() {
       reading = 0;
   }

   public TicketCounter6(int reading) {
     this.reading = reading % LIMIT;
   }

   // Increase the reading by one
   public void increase() {
     reading = (++reading) % LIMIT;
   }

   // Increase the reading by the specified amount
   public void increase(int amount) {
     reading = (reading + amount) % LIMIT;
   }

   // Retrieve the value of the attribute reading
   public int getReading() {
     return reading;
   }

   // Check whether two objects are equal
   public boolean equals(Object obj) {
       if (! (obj instanceof TicketCounter6)) {
           return false;
       }
       TicketCounter6 counter = (TicketCounter6) obj;
       return reading == counter.reading;
   }
}
```

Figure 7.95 TicketCounter6.java

Based on the `TestEquals1` class, another class `TestEquals2` is written to test the `TicketCounter6` class as shown in Figure 7.96.

```
// Definition of class TestEquals2
public class TestEquals2 {

    // Main executive method
    public static void main(String args[]) {
        // Create two TicketCounter6 objects and are referred
        // by variable counter1 and counter2
        TicketCounter6 counter1 = new TicketCounter6(10);
        TicketCounter6 counter2 = new TicketCounter6(10);

        // Show the results of == and equals() method
        System.out.println(counter1 == counter2);
        System.out.println(counter1.equals(counter2));
        System.out.println(counter2.equals(counter1));
    }
}
```

Figure 7.96 TestEquals2.java

Like the `main()` method defined in the `TestEquals1` class, two `TicketCounter6` objects are created and are referred by variables `counter1` and `counter2` respectively. Then, the results of the following three expressions are shown on the screen:

```
counter1 == counter2
counter1.equals(counter2)
counter2.equals(counter1)
```

The `==` operator simply verifies that the contents of the two variables `counter1` and `counter2` are the same. As they are referring to two different `TicketCounter6` objects, their contents must be different and the expression is evaluated to be `false`.

When the second expression is evaluated, the two `TicketCounter6` objects can be visualized in Figure 7.97.

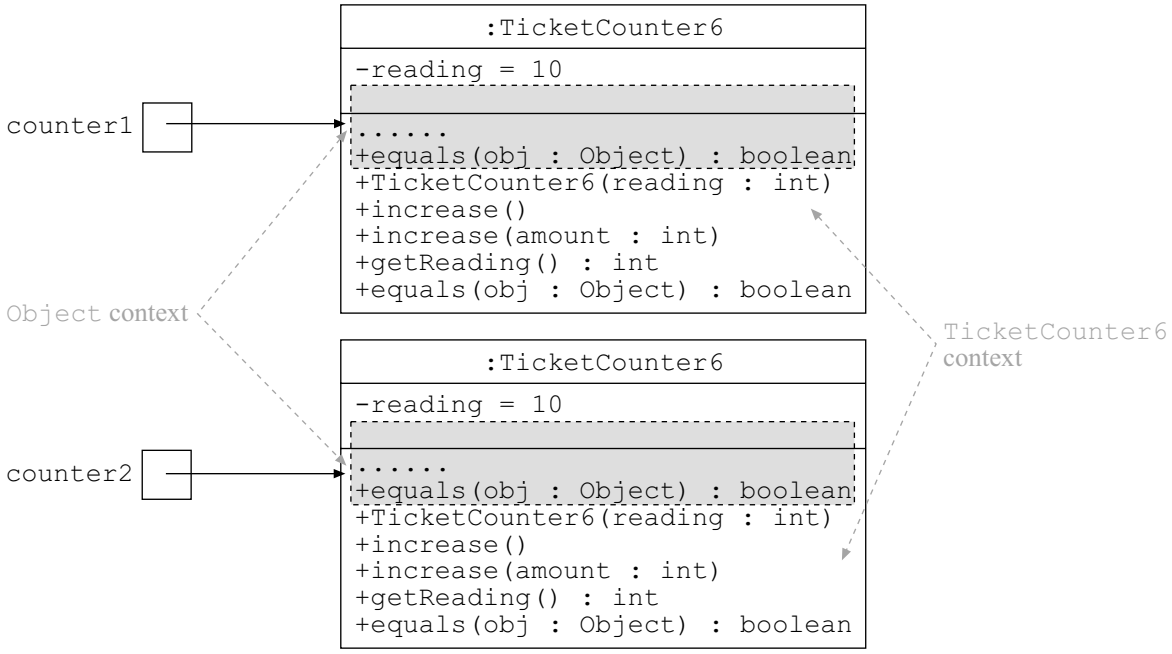


Figure 7.97 The two created `TicketCounter6` objects referred to by variables `counter1` and `counter2`

The statement `counter1.equals(counter2)` sends an `equals` message to the `TicketCounter6` object that is referred to by the variable `counter1` with supplementary data being the contents of the variable `counter2`, the reference of the second `TicketCounter6` object. Therefore, when the `equals()` method is to be executed, the scenario is as shown in Figure 7.98.

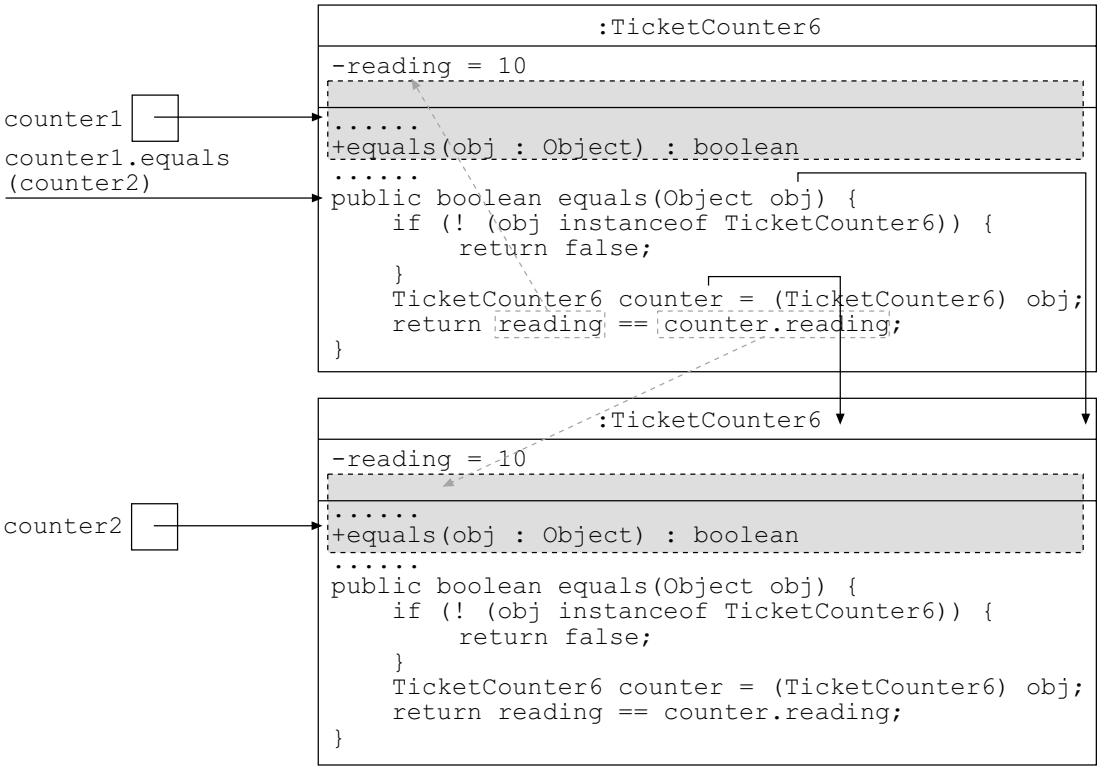


Figure 7.98 The scenario while executing the statement `counter1.equals(counter2)`

When the `TicketCounter6` object executes its `equals()` method, the `equals()` method defined in the `TicketCounter6` class overrides the one defined in the `Object` class and is executed. The parameter `obj` is referring to the second `TicketCounter6` object.

With respect to the `equals()` method defined in the `TicketCounter6`,

```
// Check whether two objects are equal
public boolean equals(Object obj) {
    if (! (obj instanceof TicketCounter6)) {
        return false;
    }
    TicketCounter6 counter = (TicketCounter6) obj;
    return reading == counter.reading;
}
```

the expression `obj instanceof TicketCounter6` is evaluated to determine whether the variable `obj` is referring to a `TicketCounter6` object. The result is `true`, as the variable `obj` is referring to a `TicketCounter6` object and the `!` operator negates the result to be `false`. The `if` part of the `if` statement is therefore skipped. Then, the content of the parameter `obj` is copied to the local variable `counter` after casting, so that the `TicketCounter6` object is referred to by a `TicketCounter6` variable. The statement is required because it is necessary to manipulate the attribute `reading` that is specific to the `TicketCounter6` class. Finally, the two `TicketCounter6` objects are checked for equality with respect to the values of their attribute `reading`. For further details of casting and the use of the `instanceof` operator, please refer to Appendix H.

As the values of the two `TicketCounter6` objects are both `10`, the expression

```
reading == counter.reading
```

is evaluated to be `true` and is returned as the return value of the method `equals()`.

For the third expression `counter2.equals(counter1)`, the scenario is identical except the object that executes its `equals()` method is the one referred by the variable `counter2`. The return value of the method call is also `true`.

Therefore, by compiling the classes `TicketCounter6` and `TestEquals2` classes and executing the `TestEquals2` class, the output is:

```
false
true
true
```

Comparing the `TicketCounter4` with the `TicketCounter6` classes, you can see that it is preferable for a class to define a suitable `equals()` method to specify the way two objects of the class are

considered to be equal. Furthermore, unless it is necessary to check whether two variables are referring to the same object, it is usually preferable to use the `equals()` method for a context-sensitive equality test instead of using the `==` operator.

Self-test 7.12

1 It is mentioned that `counter1.equals(counter2)` should always equal `counter2.equals(counter1)` where `counter1` and `counter2` are `TicketCounter6` variables. Is it always true? State when the execution results will be different.

2 Modify the class `Date1` (Figure 7.48) so that it defines its own `equals()` method. The `equals()` method returns `true` if two `Date1` objects are considered equal if the values of their attributes `day`, `month` and `year` are the same.

3 With respect to the modified `Date1` class with the `equals()` method mentioned in Question 2, what will be the output of the following program segment

```
Date1 date1 = new Date1();
date1.setDay(1);
date1.setMonth(2);
date1.setYear(2003);
BritishDate1 date2 = new BritishDate1();
date2.setDay(1);
date2.setMonth(2);
date2.setYear(2003);
System.out.println(date1.equals(date2));
System.out.println(date2.equals(date1));
```

in which `BritishDate1` is a subclass of the `Date1` class obtained in Question 2 and the `BritishDate1` class is defined as shown in Figure 7.50?

4 Based on Question 3 above (or otherwise), please comment on whether it is necessary to define an `equals()` method for subclasses of the `Date1` class.

5 For any non-primitive variable, say `npVar`, discuss the result of the expression `npVar.equals(npVar)`?

As mentioned in the reading, the `toString()` method is another frequently used method that is defined in the `Object` class and is inherited by all classes in the Java programming language. Usually, the method is used implicitly. For example, there are overloaded `System.out. print()` methods and `System.out.println()` methods that can accept all primitive types and three common non-primitive types, which are

```
public void print(char[] s) { ..... }
public void print(Object obj) { ..... }
public void print(String s) { ..... }
```

and

```
public void println(char[] s) { ..... }
public void println(Object obj) { ..... }
public void println(String s) { ..... }
```

(The package `java.lang` is usually understood, and it is therefore not shown for classes `Object` and `String` in the above method definitions.)

We discussed the `saySomethingAboutDate()` method defined in the `DateHandler1` class (Figure 7.58) earlier, which illustrates a method that can accept an object of a superclass (the general type). It can accept an object of any its subclasses (the specific types). Therefore, if the reference of an object of type other than `char[]` and `String` is supplied to the `System.out.print()` or `System.out.println()` method, the method with a parameter list of an `Object` type is chosen for execution. The reason is that all classes are subclasses of the `Object` class, and an object supplied to the methods can be considered to be an `Object` class object, and the method that accepts an `Object` class object is chosen for execution.

When a `System.out.print()` method or a `System.out.println()` method that accepts an `Object` class object is executing, a message `toString` is sent to the supplied object so that the object executes its `toString()` method and returns a `String` object as a textual representation of the object. Such textual representation is then shown on the screen. As a `toString()` method is defined in the ultimate superclass `Object`, this method is used if the class or all of its superclasses do not define an overriding `toString()` method.

For example, the `TestToString1` class as shown in Figure 7.99 shows a `Date1` object on the screen with the use of the implicitly inherited `toString()`.

```
// Definition of class TestToString1
public class TestToString1 {

    // Main executive method
    public static void main(String args[]) {
        // Create a Date1 object
        Date1 date = new Date1();
        // Set the object attributes
        date.setDay(1);
        date.setMonth(2);
        date.setYear(2003);
        // Show the Date1 object and the toString() method
        // is implicitly called
        System.out.println(date);
    }
}
```

Figure 7.99 TestToString1.java

Compile and execute the `TestToString1` class. It shows something look like

```
Date1@1a16869
```

on the screen. The output of the program may vary if you run it on your own machine. The significance is that the default textual representation, determined by the `toString()` method defined in the `Object` class, is of little use to us. Therefore, we can define an overriding `toString()` for the `Date1` class. It is a good idea that it involves the object attributes `day`, `month` and `year` in the textual representation. Based on the `Date1` class, a class named `Date3` is defined as shown in Figure 7.100.

```
// definition of the class Date3
public class Date3 {
    // Attributes
    protected int day;        // the day
    protected int month;      // the month
    protected int year;       // the year

    // Getter methods

    // get the day
    public int getDay() {
        return day;
    }

    // get the month
    public int getMonth() {
        return month;
    }

    // get the year
    public int getYear() {
        return year;
    }

    // Setter methods

    // set the day
    public void setDay(int day) {
        this.day = day;
    }

    // set the month
    public void setMonth(int month) {
        this.month = month;
    }

    // set the year
    public void setYear(int year) {
        this.year = year;
    }
```

```
    // Return a textual representation of the object
    public String toString() {
        return
            year +
            ((month < 10) ? "0" + month : "" + month) +
            ((day < 10) ? "0" + day : "" + day);
    }
}
```

Figure 7.100 Date3.java

To test the `toString()` method of the `Date3` class, a class `TestToString2` is written in Figure 7.101.

```
// Definition of class TestToString2
public class TestToString2 {

    // Main executive method
    public static void main(String args[]) {
        // Create a Date3 object
        Date3 date = new Date3();
        // Set the object attributes
        date.setDay(1);
        date.setMonth(2);
        date.setYear(2003);
        // Show the Date2 object and the toString() method
        // is implicitly called
        System.out.println(date);
    }
}
```

Figure 7.101 TestToString2.java

Compile the classes and execute the `TestToString2` class. The output to be shown on the screen is:

```
20030201
```

You can see that the `toString()` method defined in the `Date3` class is implicitly called in the `System.out.println()` method. Therefore, you can consider that

```
System.out.println(obj);
```

is equivalent to:

```
System.out.println(obj.toString());
```

The `toString()` method provides a handy way to debug your program. Whenever you want to know the object attributes of an object, you can simply define a suitable `toString()` method in the class for constructing and returning a `String` object that includes the necessary object attributes. Then, whenever you want to show the object attributes,

you insert a `System.out.println()` method (or `System.out.print()` method) with the object reference as a parameter value to show the object attributes, such as:

```
Date3 date = new Date3();
...... // Manipulate the Date3 object
System.out.println(date);
...... // Further manipulate the Date3 object
System.out.println(date);
```

If you want to change the way the object attributes show, you can modify a single `toString()` method. Otherwise, you would have to insert statements like

```
Date1 date = new Date1();
......
System.out.println(
    date.getYear() + "-" +
    date.getMonth() + "-" +
    date.getDay());
......
System.out.println(
    date.getYear() + "-" +
    date.getMonth() + "-" +
    date.getDay());
```

in your program. Then, whenever you want to modify the format for showing the object attributes, you have to search all the class definitions for the statements showing the object attributes and amend them accordingly, which is a tedious task.

The `toString()` method is also frequently used implicitly in another situation — string concatenation. You learned that the `+` operator can be used with `String` objects on either one side or both sides of the operator. For example, the statements

```
System.out.println("1 + 2 = " + (1 + 2));
System.out.println("Hello " + "World");
```

will show the following outputs on the screen:

```
1 + 2 = 3
Hello World
```

If either side of the `+` operator is a `String` object and another side is any non-primitive type, the `toString()` method of the non-primitive type object is called implicitly. That is, the statements

```
Date3 date = new Date3();
......
String message = "date=" + date;
```

can be considered to be:

```
Date3 date = new Date3();
......
String message = "date=" + date.toString();
```

The `toString()` method of the `Date3` object is called to obtain a textual representation to be concatenated with the `String` object on the left-hand side of the `+` operator.

To illustrate that the `toString()` method is called implicitly in a string concatenation operation, a class `TestToString3` is written in Figure 7.102.

```
// Definition of class TestToString3
public class TestToString3 {

    // Main executive method
    public static void main(String args[]) {
        // Create a Date1 object
        Date1 date1 = new Date1();
        date1.setDay(1);
        date1.setMonth(2);
        date1.setYear(2003);
        // Create a Date1 object
        Date3 date2 = new Date3();
        date2.setDay(1);
        date2.setMonth(2);
        date2.setYear(2003);
        // Show the Date1 object and the toString() method
        // is implicitly called
        System.out.println("date1=" + date1);
        System.out.println("date2=" + date2);
    }
}
```

Figure 7.102 TestToString3.java

Compile and execute the `TestToString3` class. The following output will be shown.

```
date1=Date1@17182c1
date2=20030201
```

The `toString()` method executed by the `Date1` object is the one defined in the `Object` class; the one executed by the `Date3` object is defined in the `Date3` class. Please be reminded that the textual representation of the `Date1` object may vary when you run the program in your own machine.

Therefore, an even better way to debug your program is to insert statements that look like the following in your program:

```
System.out.println("DEBUG:date1=" + date1);
```

We can now derive classes `AmericanDate3` and `BritishDate3` as subclasses of the `Date3` class, as shown in Figure 7.103 and Figure

7.104. They define their own method `toString()`. Their `toString()` methods override the one defined in the `Date3` class.

```
// Definition of class AmericanDate3
public class AmericanDate3 extends Date1 {

    // Methods

    // Return a textual representation of the object
    public String toString() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Return the date in m/d/yyyy format as String
        return month + "/" + day + "/" + year;
    }
}
```

Figure 7.103 AmericanDate3.java

```
// Definition of class BritishDate3
public class BritishDate3 extends Date1 {

   // Methods

   // Return a textual representation of the object
   public String toString() {
     // Get the day, month and year
     int day = getDay();
     int month = getMonth();
     int year = getYear();
     // Return the date in dd/mm/yyyy format as String
     return
       ((day < 10) ? "0" + day : "" + day) +
       "/" +
       ((month < 10) ? "0" + month : "" + month) +
       "/" +
       year;
   }
}
```

Figure 7.104 BritishDate3.java

To test the `AmericanDate3` and `BritishDate3` class, a class `TestToString4` is written in Figure 7.105.

```
// Definition of class TestToString4
public class TestToString4 {

    // Main executive method
    public static void main(String args[]) {
        // Create a AmericanDate3 object
        AmericanDate3 date1 = new AmericanDate3();
        date1.setDay(1);
        date1.setMonth(2);
        date1.setYear(2004);
        // Create a BritishDate3 object
        BritishDate3 date2 = new BritishDate3();
        date2.setDay(1);
        date2.setMonth(2);
        date2.setYear(2004);
        // Show the objects and the toString() methods
        // are implicitly called
        System.out.println("date1=" + date1);
        System.out.println("date2=" + date2);
    }
}
```

Figure 7.105 TestToString4.java

Compile the classes and execute the `TestToString4` class. It shows the following outputs on the screen, which illustrate the `toString()` methods specific to the classes, are called implicitly.

```
date1=2/1/2004
date2=01/02/2004
```

Please use the following self-test to test your understanding of the `toString()` method.

Self-test 7.13

1 The reading in this appendix suggests that the object's `toString()` method is called if the `System.out.print()` or `System.out.println()` methods are used to print an object. That is, the statement `System.out.println(obj)` and `System.out.println(obj.toString())` are practically equivalent, where `obj` is a variable of a non-primitive type. Discuss whether it is always true, and suggest a scenario if the execution results are different.

2 Modify the `TicketCounter6` class by adding a `toString()` method to it, so that it returns a `String` object with the content looking like:

```
reading=100
```

Also, write a `TestToString` class to create a `TicketCounter6` object, and use the `System.out.println()` method to show the `TicketCounter6` object.

Appendix C: implicit casting involved in method calls

We said that the `println()` method that we usually use is overloaded and the class that defines the `println()` method provides six `println()` methods that cover six of the eight primitive types in the Java programming language. How about the remaining two, `byte` and `short`? For example, will the following statement cause a compilation error?

```
byte b = (byte) 65;
System.out.println(b);
```

The answer is 'no'. When the compiler software compiles the above program segment, it determines that the type of the parameter to be supplied to the `println()` method is `byte`, but there is no corresponding method. Then, the compiler software will try to convert the data type to its 'nearest' type so that a suitable method is found. The nearest type is determined by following the arrows in the diagram in Figure 7.106 that you came across in *Unit 3*.

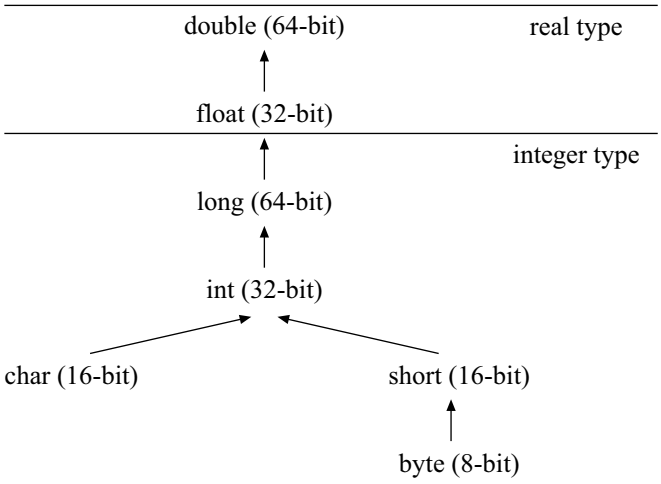


Figure 7.106 The path of implicit type conversion

As a result, the data are implicitly converted from type `byte` to `int` and the method

```
public void println(int x) { ... }
```

is used here.

We said that a method call might involve implicit-type conversion. If the method to be called is overloaded — that is, two methods defined in the class are of the same name but with different parameter lists — the implicit type conversion may cause ambiguity in choosing the method to be called. For example, the `findSum()` methods defined in the class `SumFinder` as shown in Figure 7.107 may cause such a problem.

```
// Definition of class SumFinder
public class SumFinder {
    // Determine the sum of a long and an integer
    public static long findSum(long num1, int num2) {
        return num1 + num2;
    }

    // Determine the sum of an integer and a long
    public static long findSum(int num1, long num2) {
        return num1 + num2;
    }
}
```

Figure 7.107 SumFinder.java

The two overloaded `findSum()` methods determine the sum of two parameters of types `long` and `int`, and each method accepts a different order of parameter types. What if the types of two data items supplied to the method are both `int`? To illustrate such a problem, a testing class `TestSumFinder` is written in Figure 7.108.

```
// Definition of TestSumFinder
public class TestSumFinder {

    // Main executive method
    public static void main(String args[]) {
        System.out.println(SumFinder.findSum(1L, 2));
        System.out.println(SumFinder.findSum(2, 3L));
        System.out.println(SumFinder.findSum(1, 2));
    }
}
```

Figure 7.108 TestSumFinder.java

Compile the `TestSumFinder` class. The following message is obtained:

```
TestSumFinder.java:8: reference to findSum is ambiguous, both method
findSum(long,int) in SumFinder and method findSum(int,long) in SumFinder
match
        System.out.println(SumFinder.findSum(1, 2));
                                     ^
1 error
```

The compilation error message clearly states the call of the `findSum()` method with two values of type `int` can cause ambiguity. The reason is that implicitly converting either one parameter from type `int` to `long` matches a particular `findSum()` method. The compiler software therefore cannot make a decision and a compile-time error occurs.

The aforementioned situation is the problem of ambiguity due to the implicit conversion of primitive type data. It is also a possible problem

for non-primitive type data. A class `DateStringPrinter` with overloaded `print()` methods is defined as shown in Figure 7.109.

```
// Resolve the Date class
import java.util.*;

// Definition of class DateStringPrinter
public class DateStringPrinter {
    // Show a date with a prompt
    public static void print(Date date) {
        System.out.println("DATE: " + date);
    }

    // Show a string with a prompt
    public static void print(String string) {
        System.out.println("STRING: " + string);
    }
```

Figure 7.109 DateStringPrinter.java

The `DateStringPrinter` class defines two `print()` methods with different parameter lists. The first one accepts a reference to a `Date` object, and the second one accepts a reference to a `String` object. The compiler software can determine the method to be used if either a reference to a `Date` object or a `String` object is supplied. However, the use of `null` may cause compilation error, as in the class `TestDateStringPrinter` shown in Figure 7.110.

```
// Resolve the Data class
import java.util.*;

// Definition of class TestDateStringPrinter
public class TestDateStringPrinter {

    // Main executive method
    public static void main(String args[]) {
        DateStringPrinter.print(new Date());
        DateStringPrinter.print("A String");
        DateStringPrinter.print(null);
    }
```

Figure 7.110 TestDateStringPrinter.java

Compile the `TestDateStringPrinter` and the following message is obtained:

```
TestDateStringPrinter.java:11: reference to print
is ambiguous, both method print(java.util.Date) in
DateStringPrinter and method
print(java.lang.String) in DateStringPrinter match
     DateStringPrinter.print(null);
                       ^
1 error
```

The above compilation error suggests that the value `null` can be treated as a reference to a `Date` object or a reference to a `String` object. More exactly, a `null` value can be treated as a possible value for any non-primitive parameter. Therefore, the compiler software cannot choose the one to be used, and it shows the compilation error.

Appendix D: an alternative approach for writing overloaded constructors

The execution of constructors in the `Item1` class (Figure 7.34) starts from the one with fewer parameters and goes to the one with more parameters. The sequence of constructor execution can be reversed so that a constructor with more parameters calls another constructor with fewer parameters. The constructors of the class `Item2` are defined in this way in Figure 7.111.

```
// Definition of class Item2
public class Item2 {
    // Default values
    private static final String DEFAULT_NAME = "Untitled";
    private static final double DEFAULT_PRICE = 0.0;
    private static final int DEFAULT_QUANTITY = 1;

    // Attributes
    private String name;
    private double price;
    private int quantity;

    // Behaviours

    // The constructors of the class
    public Item2() {
        // Set all attributes to default values
        name = DEFAULT_NAME;
        price = DEFAULT_PRICE;
        quantity = DEFAULT_QUANTITY;
    }

    public Item2(String theName) {
        // Call the constructor with empty parameter list to
        // initialize all attributes
        this();
        // Override the attribute name
        name = theName;
    }

    public Item2(String theName, double thePrice) {
        // Call the constructor with the name to initialize the
        // attribute name
        this(theName);
        // Override the attribute price
        price = thePrice;
    }

    public Item2(String theName, double thePrice, int theQuantity){
```

```
        // Call the constructor with the name and price to initialize
        // the attribute name and price
        this(theName, thePrice);
        // Override the attribute quantity
                    quantity = theQuantity;
                }

    // Set the attribute name
    public void setName(String theName) {
        name = theName;
    }

    // Set the attribute price
    public void setPrice(double thePrice) {
        price = thePrice;
    }

    // Set the attribute quantity
    public void setQuantity(int theQuantity) {
        quantity = theQuantity;
    }

    // Get the subtotal of this item
    public double findTotal() {
        return price * quantity;
    }
}
```

Figure 7.111 Item2.java

You can see that constructors with more parameters use the constructors with fewer parameters. Let's trace the creation process for creating an `Item2` object with the statement:

```
new Item2("Coke", 2.5, 6)
```

Executing the above statement, the JVM allocates a memory block for the `Item2` object, and all its attributes are implicitly initialized as shown in Figure 7.112.

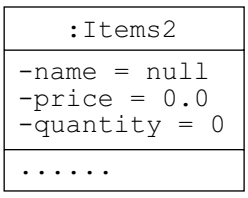
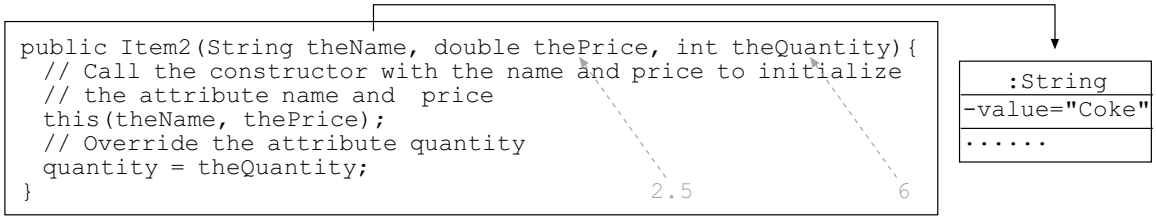


Figure 7.112 The Item2 object after implicit initialization

The next step is the explicit initialization. However, according to the attribute declarations, there is no explicit initialization.

```
private String name;
private double price;
private int quantity;
```

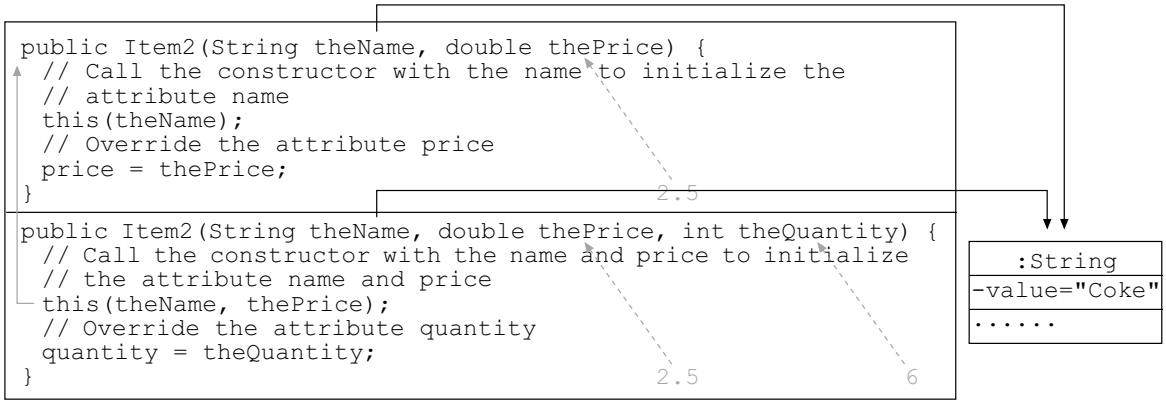
Then, the constructor with a parameter list that matches the parameter value list of the `new` statement is executed. Therefore, the constructor with a parameter list containing a `String`, a `double` and an `int` is executed.



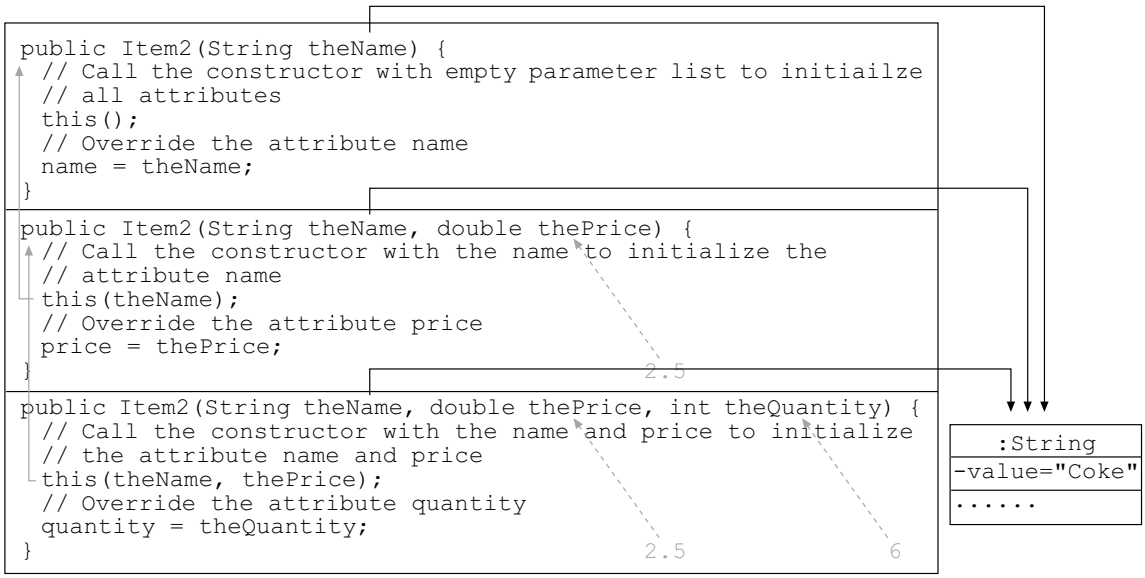
The first statement of the above constructor

```
this(theName, thePrice);
```

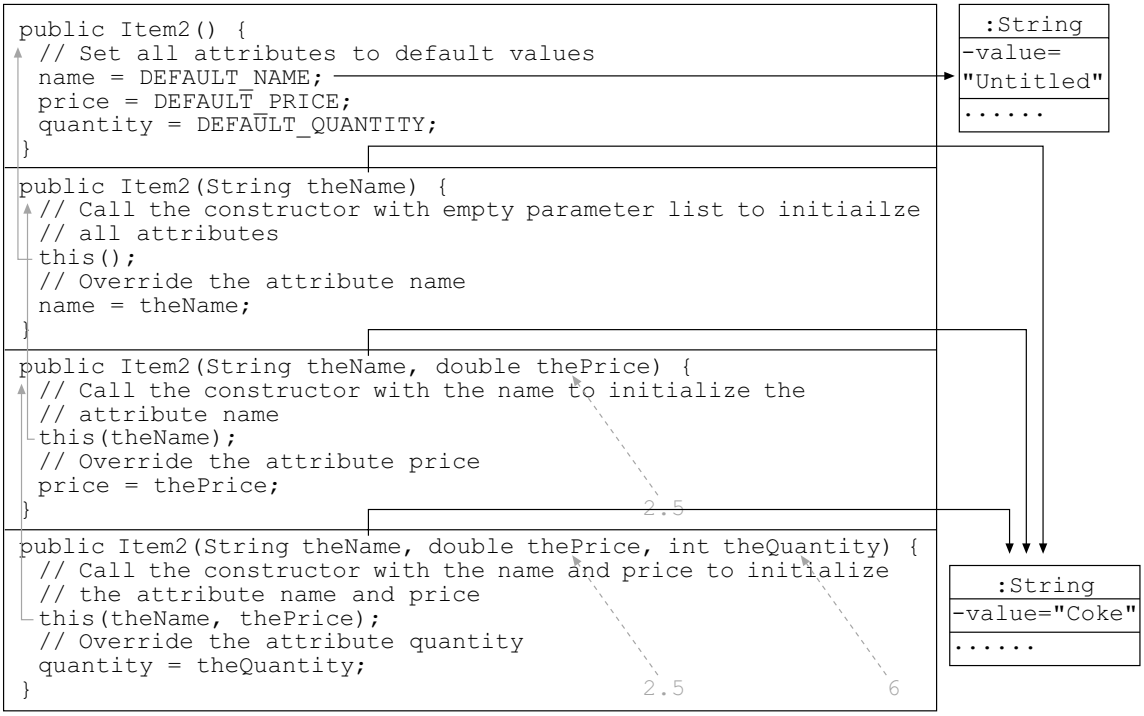
executes another constructor of the same class with a parameter list containing a `String` and a `double`.



The constructor with a parameter list of a `String` and a `double` obtains two pieces of data — `"Coke"` and `2.5`. The first statement of the constructor further calls another constructor with a parameter list of a `String`.



The first statement in the constructor with a parameter list of a `String` further calls the constructor with an empty parameter list.



The constructor with an empty parameter list initializes the object attributes with the default values declared as `final` class variables in the class. After completing the execution of the constructor, the `Item2` object is initialized as shown in Figure 7.113.

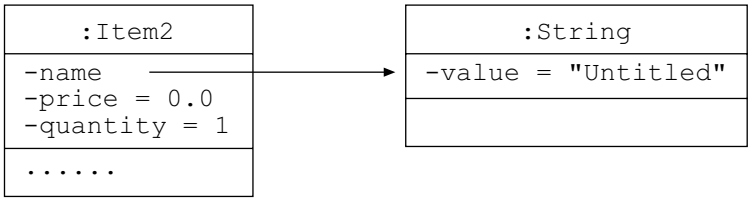
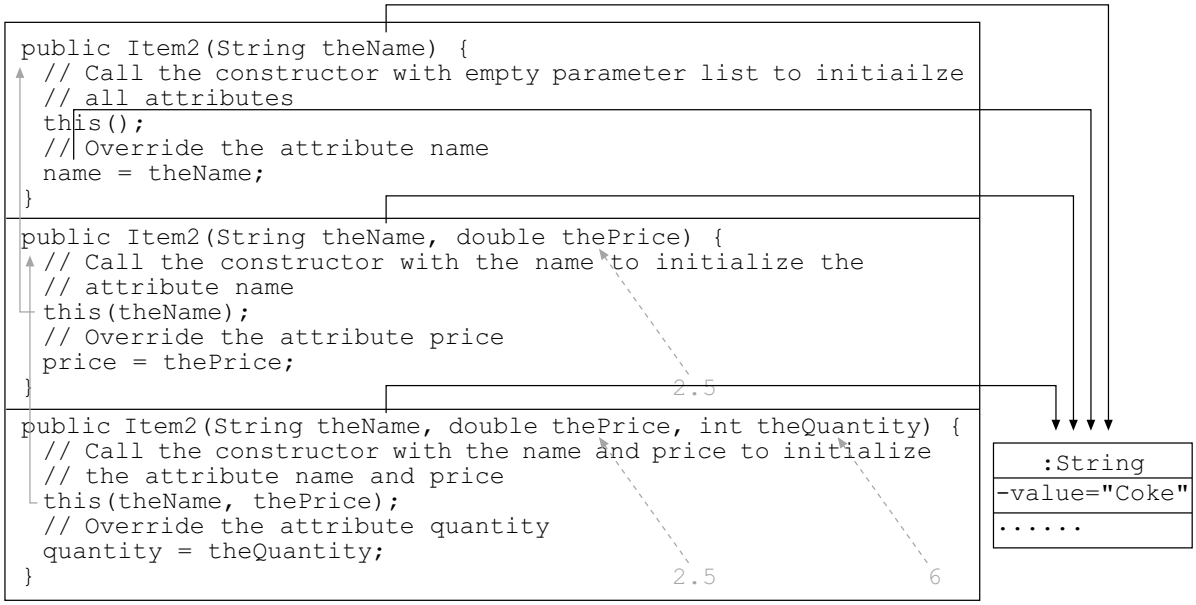


Figure 7.113 The Item2 object after executing the Item2() constructor

Afterwards, the flow of control is returned to the constructor with a parameter list of a `String`.



Then, the value of the parameter `theName` is assigned to the object attribute `name`. After completing the execution of this constructor, the `Item2` object is shown in Figure 7.114.

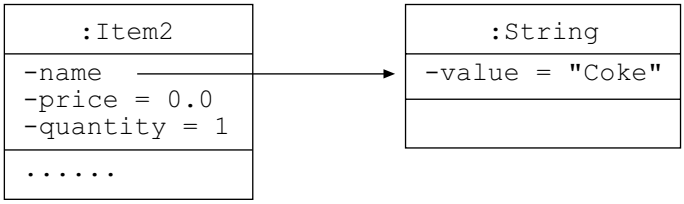
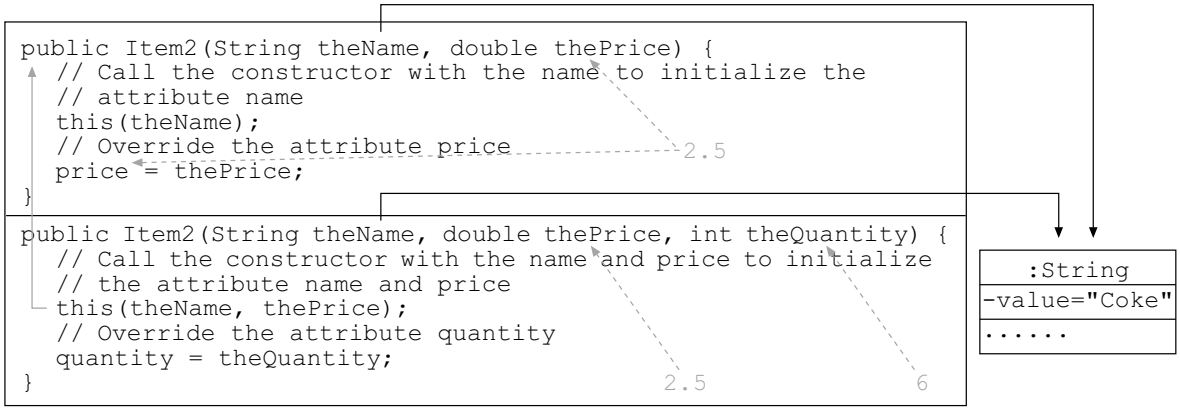


Figure 7.114 The Item2 object after executing the Item2(String) constructor

Afterwards, the flow of control is returned to the constructor with parameters of a `String` and a `double`. That is:



The value stored in the parameter `thePrice`, `2.5` is assigned to the object attribute `price`. As a result, the object after completing the execution of `Item2(String,double)` constructor becomes as shown in Figure 7.115.

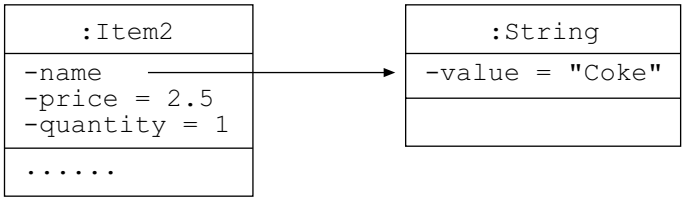
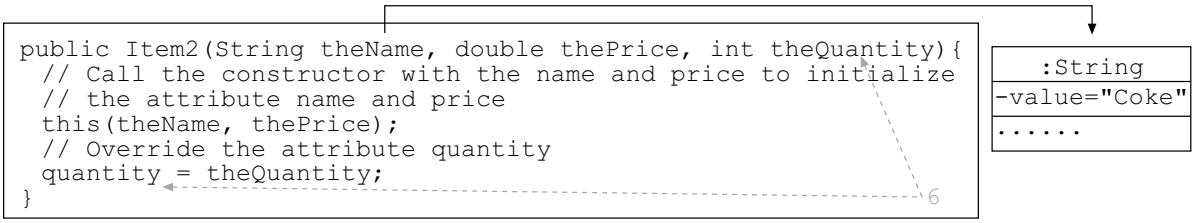


Figure 7.115 The Item2 object after executing the Item2(String, double) constructor

Finally, the flow of control is returned to the original constructor with parameters of a `String`, a `double` and an `int`.



Then, the value stored in the parameter `theQuantity` is assigned to the attribute `quantity`, and `Item2` is finally initialized to be as shown in Figure 7.116.

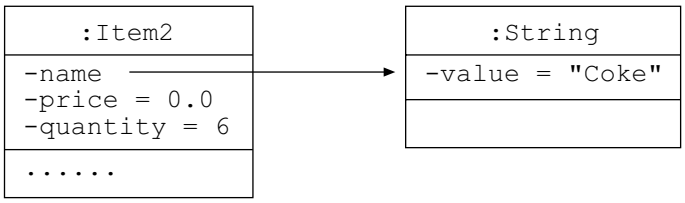


Figure 7.116 The `Item2` object after executing the `Item2(String, double, int)` constructor

Of the definitions of classes `Item1` and `Item2`, the former, `Item1`, is preferable because the operations are consolidated in a single constructor and hence there is a single point of modification. The other constructors only prepare the necessary parameter lists.

Appendix E: testing the contents of the keyword `this`

Let's verify the contents of an imaginary attribute `this`. A class `TicketCounter7` is written as shown in Figure 7.117 based on that of class `TicketCounter3`. There is a new `showThis()` method that simply calls the `println()` method with the virtual attribute `this`.

```
// Definition of class TicketCounter7
public class TicketCounter7 {
   // Attribute
   public static final int LIMIT = 1000;
   private int reading;    // The reading of the counter

   // The constructors
   public TicketCounter7() {
       reading = 0;
   }

   public TicketCounter7(int theReading) {
      reading = theReading % LIMIT;
   }

   // Increase the reading by one
   public void increase() {
      reading = (++reading) % LIMIT;
   }

   // Increase the reading by the specified amount
   public void increaseByAmount(int amount) {
      reading = (reading + amount) % LIMIT;
   }

   // Retrieve the value of the attribute reading
   public int getReading() {
      return reading;
   }

   // Show a message with "this"
   public void showThis() {
       System.out.print(
           "DEBUG: this in showThis() of TicketCounter7 = ");
       System.out.println(this);
   }
}
```

Figure 7.117 TicketCounter7.java

A testing program is written as the `TestThis` class in Figure 7.118.

```
// Definition of class TestThis
public class TestThis {

  // Main executive method
  public static void main(String args[]) {
    // Create an TicketCounter7 object
    TicketCounter7 counter = new TicketCounter7();
    // Show message with local variable counter
    System.out.print("DEBUG: counter in main() = ");
    System.out.println(counter);
    // Request the TicketCounter7 object to show a message by "this"
    counter.showThis();
  }
}
```

Figure 7.118 TestThis.java

The following statement in the `main()` method of `TestThis` class

```
TicketCounter7 counter = new TicketCounter7();
```

creates a `TicketCounter7` object. The scenario can be visualized in Figure 7.119.

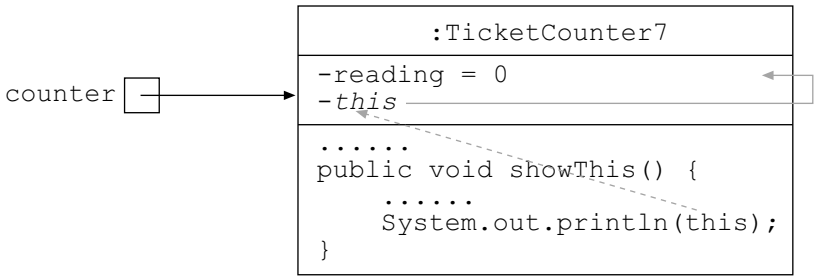
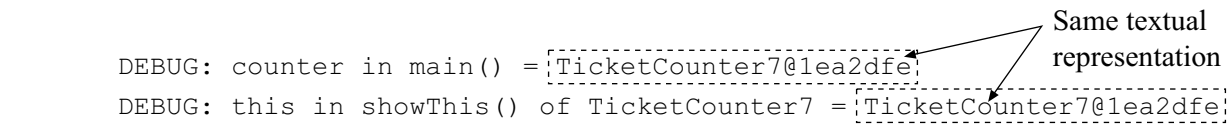


Figure 7.119 A `TicketCounter7` object with the imaginary attribute `this`

(The attribute `this` in the diagram is set in italics to highlight that it is not an attribute defined by the class but an imaginary one. Furthermore, the method body is shown to help you to visualize the scenario. In addition, the dashed arrow in the diagram shows the interpretation of a variable.)

To verify the scenario, compile and execute the `TestThis` program. The following outputs are shown on the screen:



If the `println()` method (and `print()` method) is supplied with a reference to an object, the referred object is requested to provide a textual representation as a `String` and is then shown on the screen. The default textual representation is the class name of the object, an `@` character and a reference like a hexadecimal number.

If you execute the `TestThis` program on your own machine, the hexadecimal numbers that follow the `@` characters may be different, but they are identical for the same execution. It confirms that the object referred to by the local variable `counter` in the `main()` method of `TestThis` is the same one as referred by the keyword `this` in the `showThis()` method of `TicketCounter7` object in Figure 7.119.

Appendix F: overloading methods in a subclass

If the parameter lists of the methods defined in the superclass and the subclass are different, it is overloading. For example, a class named `Timer` is designed as shown in Figure 7.120.

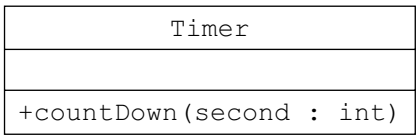


Figure 7.120 The design of the `Timer` class

Based on the design of the class `Timer`, it is possible to derive a subclass `PreciseTimer` as shown in Figure 7.121.

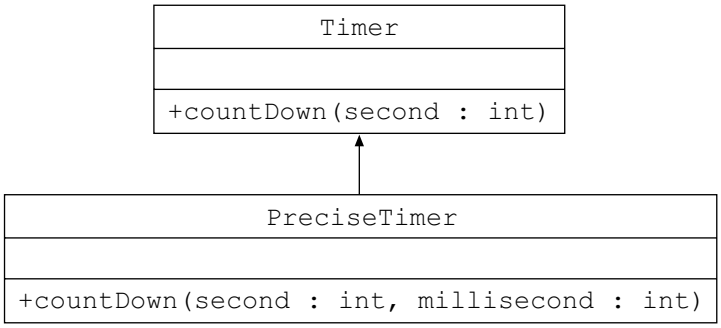


Figure 7.121 The design of the `PreciseTimer` class

The relationship between the classes `PreciseTimer` and `Timer` is that 'a `PreciseTimer` is a `Timer`'. Therefore, it makes sense to derive the `PreciseTimer` class as a subclass of the `Timer` class, as there is an 'is a' relationship between them in which `PreciseTimer` is a specific type of the general type `Timer`.

Based on the design as shown in Figure 7.121, the classes `Timer` and `PreciseTimer` can be defined as shown in Figure 7.122 and Figure 7.123 respectively.

```
// Definition of class Timer
public class Timer {
    // To wait for the countdown for the specified period
    // It is assumed that an iteration of the for loop lasts for
    // 1/1000 second
    public void countDown(int second) {
        for (int i=0; i < second * 1000; i++) {
        }
    }
}
```

Figure 7.122 Timer.java

```
// Definition of class PreciseTimer
public class PreciseTimer extends Timer {
    // To wait for the countdown for the specified period
    // It is assumed that an iteration of the for loop lasts for
    // 1/1000 second
    public void countDown(int second, int millisecond) {
        for (int i=0; i < second * 1000 + millisecond; i++) {
        }
    }
}
```

Figure 7.123 PreciseTimer.java

An alternative way to do so is to use a pair of curly braces containing no statement. The implementations of the classes assume that an iteration of the `for` loop lasts for one-thousandth of second. It is just for illustration, and a computer can complete executing such a loop in a much shorter time.

The `countDown()` method of the `Timer` accepts a parameter of type `int`, whereas the `countdown()` method of the `PreciseTimer` accepts two parameters of type `int`. We know that the `PreciseTimer` is a subclass and hence an extension of the `Timer` class. The `PreciseTimer` therefore inherits the `countDown()` method with a parameter of type `int` from the `Timer` class. As a result, the `PreciseTimer` class actually has two `countDown()` methods with different parameter lists, which means that the following program segment is valid:

```
// Create a PreciseTimer with default constructor
PreciseTimer timer = new PreciseTimer();
timer.countDown(1000);     // countdown for 1 second
timer.countDown(1000, 500);// countdown for 1.5 second
```

The first statement creates a `PreciseTimer` object, and the remaining two statements are for countdowns of one second and 1.5 second respectively. When the `PreciseTimer` object executes the `countDown` method with a parameter of type `int`, the method to be executed is the one inherited from the `Timer` class. But, for the second method call with two parameters of type `int`, the `countDown()` method defined in the `PreciseTimer` class itself is executed.

The two `countDown()` methods are two different independent methods, as they have the same name but different parameter lists. Therefore, the `countDown()` methods are considered as being overloaded.

Appendix G: access modifiers

Access modifiers — the `private` privilege

Let's investigate the way the overriding methods in the subclasses `AmericanDate1` and `BritishDate1` are written. With respect to the `print()` method of the `BritishDate1`:

```
public void print() {
  // Get the day, month and year
  int day = getDay();
  int month = getMonth();
  int year = getYear();
  // Show the date in dd/mm/yyyy format
  System.out.print((day < 10) ? "0" + day : "" + day);
  System.out.print("/");
  System.out.print((month < 10) ? "0" + month : "" + month);
  System.out.print("/");
  System.out.print(year);
}
```

The first three statements in the method obtain the values of the attributes `day`, `month` and `year`. The statement for getting the value of the attribute `day` can be made explicit as:

```
int day = this.getDay();
```

Since the `BritishDate1` is a subclass of the `Date1` class, it inherits all attributes and methods that are defined in the `Date1` class, and it is therefore possible to send a message `getDay` to itself (via the implicit `this`) for getting the value of the attribute `day`. However, why is it necessary to get the attribute values via methods calls?

The reason is that the attributes `day`, `month` and `year` are marked `private` in the `Date1` class. Marking attributes or methods `private` means that they can only be accessed by the object of the same class. Therefore, even though the `BritishDate1` class is a subclass of the `Date1` class and hence possesses the `Date1` class attributes, the methods defined in the `BritishDate1` class cannot access them. You can imagine the `BritishDate1` object is partitioned into two contexts, a `Date1` context and a `BritishDate1` context as shown in Figure 7.124. The `Date1` context includes the attributes and methods defined by the `Date1` class, and the `BritishDate1` context refers to those defined in the `BritishDate1`.

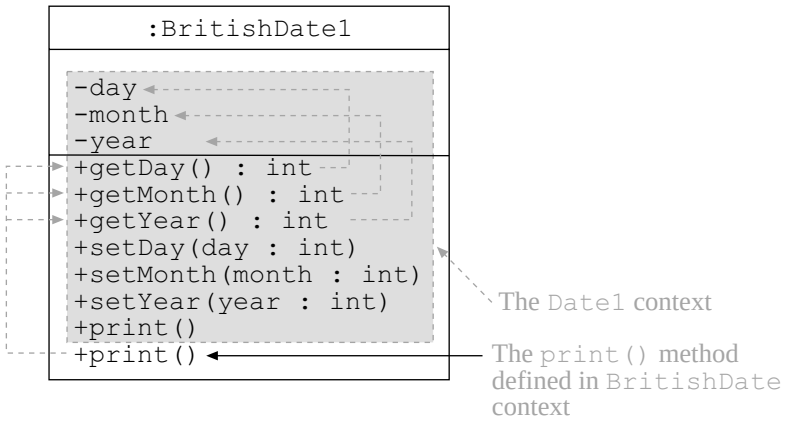


Figure 7.124 A BritishDate1 object

The shadowed part of the `BritishDate1` object shown in Figure 7.124 can be considered the `Date1` context. Whenever the `BritishDate1` object receives a message to perform a corresponding behaviour, if the method is defined in the `BritishDate1` class (and hence exists in the `BritishDate1` context of the object), it is the method to be executed. Otherwise, the method defined in the `Date1` class is executed. This can help you understand the idea that the method of the superclass is overridden if a method is defined in the subclass with an identical name and parameter list. According to this argument, the `setDay()`, `setMonth()` and `setYear()` methods to be executed by a `BritishDate1` object are those defined in the `Date1` class, whereas the `print()` method to be executed is the one defined in the `BritishDate1` class.

The members in the `Date1` context are the only ones that can access the `private` members defined in the `Date1` class (and hence within the `Date1` context of the `BritishDate1` object). The `print()` method defined by the `BritishDate1` class is in the `BritishDate1` context of the object, and it cannot access the `private` attributes `day`, `month` and `year` in the `Date1` context directly. Instead, the `print()` method can access the `public` members defined in the `Date1` context, and it can therefore call the `getDay()`, `getMonth()` and `getYear()` methods to set the attributes values. While the `getDay()` method that is defined in the `Date1` context is executing, the method can access other members defined in the `Date1` context. Therefore, the `print()` method defined in the `BritishDate1` class can access those `private` members via the `public` getter/setter methods of the `Date1` class.

In Figure 7.124 above, the dashed arrows illustrate that the method `print()` in the `BritishDate1` context accesses the `public` members `getDay()`, `getMonth()` and `getYear()` in the `Date1` context. As these three methods are in the `Date1` context, they can further access the `private` members in the `Date1` context.

As we said in the section ‘Accessing object attributes via method calls’, attributes marked `private` and methods marked `public` can realize two programming concepts, information hiding and encapsulation, which

can help programmers to maintain and reuse the class definitions. Therefore, even though the subclass needs to access the `private` attributes via calling `public` method calls, it is preferable and is recommended. For example, if the programmer of the `Date1` superclass changes the implementation, such as the name of the `private` attributes, the subclasses `BritishDate1` and `AmericanDate1` need no modification.

Access modifiers — the `protected` privilege

Occasionally, it is necessary to enable members defined in a class to be accessible directly by their subclasses. A possible way is to mark the members `public` but such an approach cannot prevent those members from being accessed directly by other objects. The Java programming language provides an access modifier — `protected` — that specifies an access privilege between `private` and `public`. Members marked `protected` in a class can be accessed by subclasses.

With respect to the `BritishDate1` class, you learned that the `BritishDate1` context of the `BritishDate1` object could access the `public` members defined in the `Date1` context, which are `public` members defined in the `Date1` superclass. Therefore, if the attributes `day`, `month` and `year` were marked as `public`, the `print()` method defined in the `BritishDate1` subclass could access them directly.

However, marking the attributes `day`, `month` and `year` as `public` violates the idea of information hiding and encapsulation, and the users of the class may be tempted to access the object attributes directly. To enable attributes and methods in the superclass to be accessible by the subclass methods, an alterative way is to mark them `protected`. For example, the `Date4` class is defined based the `Date1` class, and all attributes are marked `protected` as shown in Figure 7.125. The classes `BritishDate1` and `AmericanDate1` are modified to be classes `BritishDate4` and `AmericanDate4` that extend the `Date4` class as shown in Figure 7.126 and Figure 7.127.

```
// definition of the class Date4
public class Date4 {
    // Attributes
    protected int day;        // the day
    protected int month;      // the month
    protected int year;       // the year

    // Getter methods

    // get the day
    public int getDay() {
        return day;
    }

    // get the month
    public int getMonth() {
        return month;
```

```
    }

    // get the year
    public int getYear() {
        return year;
    }

    // Setter methods

    // set the day
    public void setDay(int day) {
        this.day = day;
    }

    // set the month
    public void setMonth(int month) {
        this.month = month;
    }

    // set the year
    public void setYear(int year) {
        this.year = year;
    }
}
```

Figure 7.125 Date4.java

```
// Definition of class BritishDate4
public class BritishDate4 extends Date4 {

    // Override the print() method defined in the Date4 class
    // and show the date in dd/mm/yyyy format
    public void print() {
        // Show the date in dd/mm/yyyy format
        System.out.print((day < 10) ? "0" + day : "" + day);
        System.out.print("/");
        System.out.print((month < 10) ? "0" + month : "" + month);
        System.out.print("/");
        System.out.print(year);
    }
}
```

Figure 7.126 BritishDate4.java

```
// Definition of class AmericanDate4
public class AmericanDate4 extends Date4 {

    // Methods

    // Override the print() method defined in the Date4 class
    // and show the date in m/d/yyyy format
    public void print() {
        // Show the date in m/d/yyyy format
        System.out.print(month + "/" + day + "/" + year);
    }
}
```

Figure 7.127 AmericanDate4.java

The relationships among the `Date4`, `AmericanDate4` and `BritishDate4` are visualized as shown in Figure 7.128.

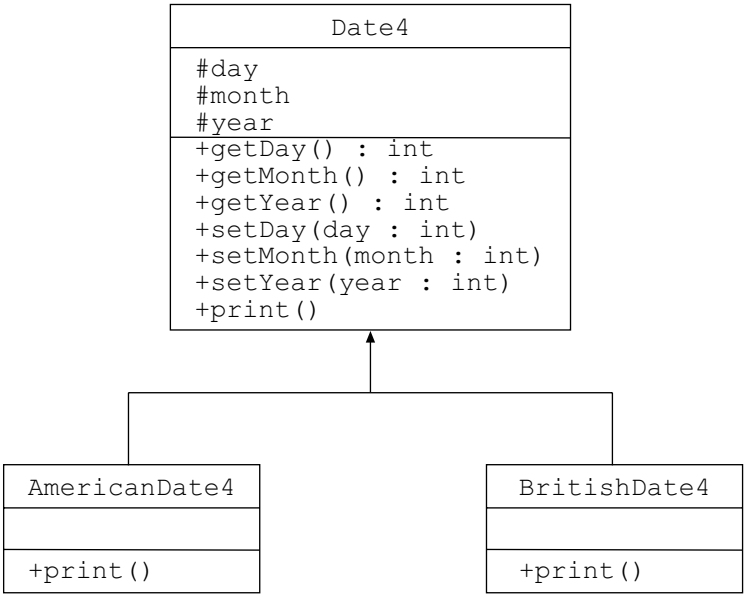


Figure 7.128 The relationships among `Date4`, `AmericanDate4` and `BritishDate4`

In Figure 7.128, the attributes `day`, `month` and `year` are prefixed with # characters to indicate that the attributes are `protected`. A `BritishDate4` object is shown in Figure 7.129.

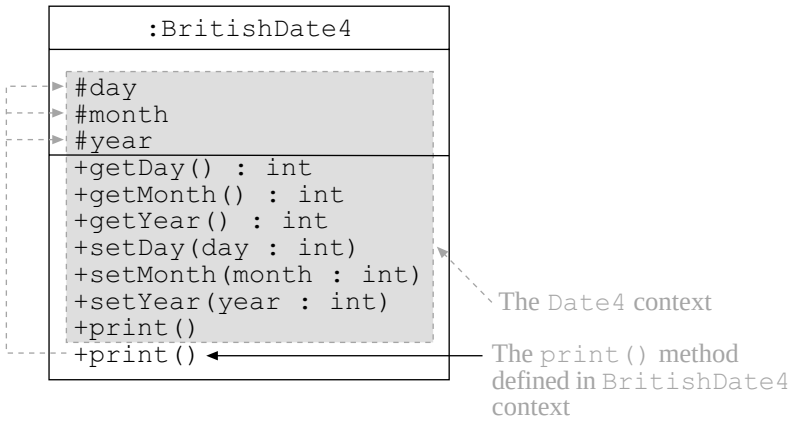


Figure 7.129 A `BritishDate4` object

The `print()` method defined in the `BritishDate4` class implies that there is a `print()` method defined in the `BritishDate4` context. Whenever the `BritishDate4` object receives a message `print`, the `print()` method in the `BritishDate4` context is executed. Since the attributes `day`, `month` and `year` in the `Date4` contexts are marked `protected`, the `print()` method in the `BritishDate4` can access the attributes directly.

Also, members marked as `protected` can be accessed by the classes in the same package. For example, if two classes, `ClassA` and `ClassB`, are in the same package `mypackage`, `ClassB` objects can access members defined in `ClassA` as being `protected`. Similarly, `protected` members of `ClassB` can be accessed by `ClassA` objects.

Access modifier — the package privilege

We have three access modifiers that govern the accessibility of object members. They are `public`, `protected` and `private`. In addition, attributes and methods can be declared without any one of the three afore-mentioned keywords. The accessibility of an attribute or method that is declared with no access modifier is considered to be *default* or *package*.

Members declared without an access modifier can be accessed by objects in the same package. The difference between the default and `protected` accessibilities is that `protected` members can be accessed by the subclasses even if the packages of the superclass and subclasses are different.

To illustrate the different accessibilities, a few classes `generic.Date`, `generic.TestDate`, `british.BritishDate` and `british.TestBritishDate` are written as shown in Figure 7.130, Figure 7.131, Figure 7.132 and Figure 7.133 respectively. These class source files are stored in the folders `generic` and `british` respectively in the CD-ROM.

```
package generic;

// definition of the class Date
public class Date {
    // Attributes
    protected int day;        // the day
    protected int month;      // the month
    protected int year;       // the year
    final int VERSION = 1;    // the version of Date definition

    // Getter methods

    // get the day
    public int getDay() {
        return day;
    }

    // get the month
    public int getMonth() {
        return month;
    }

    // get the year
    public int getYear() {
        return year;
    }

    // Setter methods

    // set the day
    public void setDay(int day) {
        this.day = day;
    }

    // set the month
    public void setMonth(int month) {
        this.month = month;
    }

    // set the year
    public void setYear(int year) {
        this.year = year;
    }

    public void print() {
        System.out.print(year);
        System.out.print((month < 10) ? "0" + month : "" + month);
        System.out.print((day < 10) ? "0" + day : "" + day);
    }
}
```

Figure 7.130 generic/Date.java

```
package generic;

// Definition of class TestDate
public class TestDate {
    public static void main(String args[]) {
        Date date = new Date();
        System.out.println("Version of Date class is " + date.VERSION);
        date.setYear(2003);
        date.setMonth(1);
        date.setDay(2);
        date.print();
    }
}
```

Figure 7.131 generic/TestDate.java

```
package british;

import generic.Date;

// Definition of class BritishDate
public class BritishDate extends Date {
    public void print() {
        // Show the date in dd/mm/yyyy format
        System.out.print((day < 10) ? "0" + day : "" + day);
        System.out.print("/");
        System.out.print((month < 10) ? "0" + month : "" + month);
        System.out.print("/");
        System.out.print(year);
    }
}
```

Figure 7.132 british/BritishDate.java

```
package british;

// Definition of class TestBritishDate
public class TestBritishDate {
    public static void main(String args[]) {
        BritishDate date = new BritishDate();
        date.setYear(2003);
        date.setMonth(1);
        date.setDay(2);
        date.print();
    }
}
```

Figure 7.133 british/TestBritishDate.java

For the definition of class `generic.TestDate`, when the compiler software resolves the class `Date`, it automatically determines whether a `Date` class is defined in the same package, that is, `generic.Date`.

Therefore, it is not necessary to provide an `import` statement for the `generic.Date` class. Similarly, for the class definition of `british.TestBritishDate`, the compiler software implicitly determines whether there is a `BritishDate` in the same package, which is `british.BritishDate`. Then an `import` statement for `british.BritishDate` is optional. As the parent class of `british.BritishDate` is `generic.Date`, which is not in the same package, it is necessary to resolve the `generic.Date` by an `import` statement, or the definition of the `british.BritishDate` can be modified to be:

```
package british;

// Definition of class BritishDate
public class BritishDate extends generic.Date {
    ......
}
```

Figure 7.134 shows the relationship among the four classes and the accessibilities.

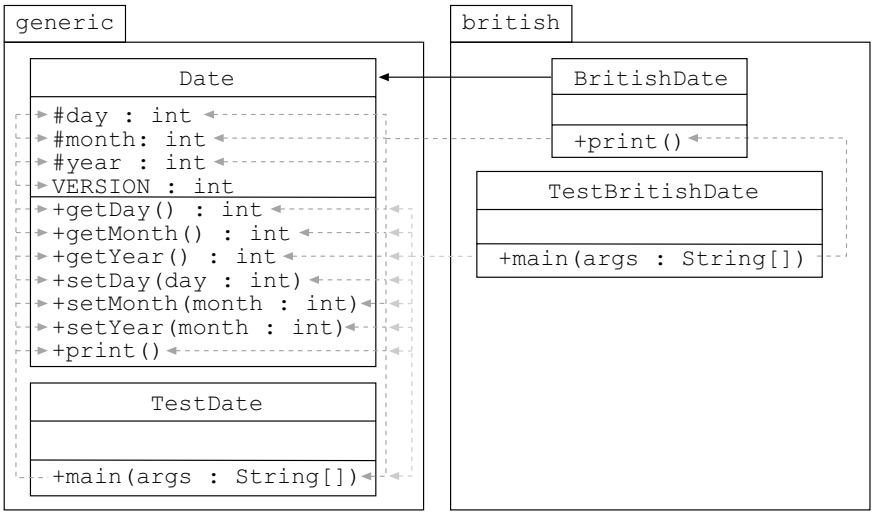


Figure 7.134 The four classes in two different packages

(The class members, which are `static` attribute and methods, are underlined to allow for differentiation. The member with package accessibility is shown without any prefix.)

The hashed arrows in Figure 7.134 show which classes can access a member. The `TestDate` class can access `Date` class members that are marked to be `protected`, `public` and *default* (that is, no access modifier). As `BritishDate` class is defined as a subclass of `Date` but they are not in the same package, `BritishDate` class can only access `protected` members of the `Date` class and `public` members of all classes in the `generic` package. The `TestBritishDate` class can only access all `public` members of classes in the `generic` package. Moreover, it can access `public`, `protected` and those without access modifier in the same package. Table 7.2 summarizes the accessibilities of different access modifiers.

Table 7.2 Accessibility of different access modifiers

	`public`	`protected`	*package*	`Private`
Same class	✓	✓	✓	✓
Same package	✓	✓	✓	×
Subclass	✓	✓	×	×
Anywhere	✓	×	×	×

Appendix H: casting and the `instanceof` operator

We have discussed the classes `Date1`, `BritishDate1` and `AmericanDate1` quite often throughout this unit. They all define a `print()` method that shows the date in a particular format. Furthermore, a date can be shown in long British and American styles. As showing a date in a long format is specific to the `BritishDate1` and `AmericanDate1` classes, we can modify the two classes to be `BritishDate5` and `AmericanDate5` as shown in Figure 7.135 and Figure 7.136.

```
// Definition of class BritishDate5
public class BritishDate5 extends Date1 {

    // Attributes
    public static final String DATE_NAME = "British";

    // An array for storing all month names
    private static final String[] MONTH_NAMES = {
        "January", "February", "March", "April",
        "May", "June", "July", "August",
        "September", "October", "November", "December"
    };

    // Override the print() method defined in the Date1 class
    // and show the date in dd/mm/yyyy format
    public void print() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in dd/mm/yyyy format
        System.out.print((day < 10) ? "0" + day : "" + day);
        System.out.print("/");
        System.out.print((month < 10) ? "0" + month : "" + month);
        System.out.print("/");
        System.out.print(year);
    }

    // Print the date in long format
    public void printLongDate() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in long format
        System.out.print(day + " ");
        System.out.print(MONTH_NAMES[month - 1] + " ");
        System.out.print(year + ".");
    }
}
```

Figure 7.135 BritishDate5.java

```
// Definition of class AmericanDate5
public class AmericanDate5 extends Date1 {

    // Attributes
    public static final String DATE_NAME = "American";

    // An array for storing all month names
    private static final String[] MONTH_NAMES = {
        "January", "February", "March", "April",
        "May", "June", "July", "August",
        "September", "October", "November", "December"
    };

    // Override the print() method defined in the Date1 class
    // and show the date in m/d/yyyy format
    public void print() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in m/d/yyyy format
        System.out.print(month + "/" + day + "/" + year);
    }

    // Print the date in long format
    public void printLongDate() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in long format
        System.out.print(MONTH_NAMES[month - 1] + " ");
        System.out.print(day + ", ");
        System.out.print(year + ".");
    }
}
```

Figure 7.136 AmericanDate5.java

We know it is possible to use a variable of a superclass type to store a reference of a subclass object, as illustrated in the `TestBritishDate2` class shown in Figure 7.56. Then, how about the following program segment?

```
Date1 date = new BritishDate5();
date.setDay(3);
date.setMonth(4);
date.setYear(2003);
date.printLongDate();
```

The first four statements are the same as those in the `main()` method of the `TestBritishDate2` class. For the last statement,

```
date.printLongDate();
```

the message `printLongDate` is sent to the object referred to by the variable `date`. According to the first four statements in the program segment, we know that the variable `date` is referring to a `BritishDate5` object that possesses a method `printLongDate()`. Is there any problem with the statement?

The interpretation of the last statement is to send a `printLongDate` message to an object that is referred by a variable `date` of type `Date1`. However, the compiler software can only figure out that an object referred to by the variable `date` has all attributes and methods defined in the `Date1` class, because the variable type is `Date1`. Therefore, the compiler software cannot predict that the object actually referred to by the variable `date` is a `BritishDate5` object at runtime, and it gives a compile-time error stating that the `Date1` class does not have such a method.

A `BritishDate5` object cannot perform a `BritishDate5` specific method, because the message type is specific to `BritishDate5` class and cannot be sent via a variable of type `Date1`. Therefore, if the `BritishDate5` object can be referred to by a variable of type `BritishDate5`, the `BritishDate5` object can perform its specific method again. For example:

```
BritishDate5 bDate = date;
bDate.printLongDate();
```

The second statement sends the message `printLongDate` to the object referred by a variable of type `BritishDate5` and the compiler software can determine that the `BritishDate5` class defines a `printLongDate()` method and the statement is therefore acceptable and executes properly at runtime. The problem left is whether the first statement is valid.

When the compiler software compiles the first statement,

```
BritishDate5 bDate = date;
```

the statement is interpreted to copy the reference that is stored in a variable of type `Date1` to another variable of type `BritishDate5`. Even though we know that the variable `date` is referring to a `BritishDate5` object, the compiler determines that it is not always true, because the variable `date` can refer to a strictly `Date1` object that is not a `BritishDate5` object. Therefore, the compiler software will show you a compile-time error.

To ‘assure’ the compiler software that you are sure the object referred by the variable `date` is actually a `BritishDate5` object, you can perform a casting operation before copying the reference that is stored in a variable of the type `Date1` to a variable of `BritishDate5`. For example, the statement can be modified to be:

```
BritishDate5 bDate = (BritishDate5) date;
```

The pair of parentheses encloses the class name to which the reference is to be casted. The reference after casting is considered to be the reference of the class enclosed in the parentheses. Therefore, the compiler software will accept the assignment operation to copy the casted reference to the variable of the type `BritishDate5`. At runtime, the casting operation will check the type of object referred to by the variable `date`. If the object is a `BritishDate5` object, the casting operation succeeds and the reference will be copied to the variable `bDate` of type `BritishDate5`. Otherwise, a runtime error will occur and the program will terminate abnormally. Therefore, the following is the complete program segment after modification:

```
Date1 date = new BritishDate5();
date.setDay(3);
date.setMonth(4);
date.setYear(2003);
BritishDate5 bDate = (BritishDate5) date;
bDate.printLongDate();
```

The casting operation with reference of a subclass type to its superclass type is implicit, and no casting operator is required. For example, the following statement is valid in all cases, such as:

```
Date1 date = new BritishDate5();
```

The reason is that the compiler software can determine that a `BritishDate5` object must possess all attributes and methods defined in the `Date1` class, due to the inheritance relationship. A casting operator can be provided

```
Date1 date = (Date1) new BritishDate5();
```

but is always redundant.

If the reference after casting is only used once, you can use the following shorthand.

```
((BritishDate5) date).printLongDate();
```

The above statement casts the reference stored by the variable `date` to a `BritishDate5` reference first, and the message `printLongDate` is sent via the casted reference. The parentheses that enclose the expression '`(BritishDate5) date`' are mandatory, because the `.` operator and the casting operator are of the same priority and are left-associative. If the parentheses were omitted, the expression would have been considered to be

```
(BritishDate5) (date.printLongDate())
```

which is invalid, because the message `printLongDate` cannot be sent via a variable of type `Date1`, and the `printLongDate()` returns no value to be caste.

The previous program segment probably will not exist in any real program, because it is equivalent to the following program segment:

```
BritishDate5 date = new BritishDate5();
date.setDay(3);
date.setMonth(4);
date.setYear(2003);
date.printLongDate();
```

It is provided here just to illustrate the necessity of casting.

As casting an incorrect object is a runtime error, it is preferable to ensure the type of the object to be casted is correct so that the casting operation must succeed. The Java programming language provides an `instanceof` operator to determine the actual type of an object. The format of using the `instanceof` operator is

object-reference `instanceof` *class-name*

in which the *object-reference* is usually a non-primitive variable and the *class-name* is the class to be verified. If the object reference is referring to an object of the class specified by the class name, the expression is evaluated to be `true`. Otherwise, the expression is `false`. If the variable stores a `null` value, the expression is always `false`.

For example, the `instanceof` operator can be used as follows:

```
date instanceof BritishDate5
```

The expression returns `true` if the variable `date` is referring to a `BritishDate5` object. However, if the variable `date` stores a `null` value, or it is not referring to a `BritishDate5`, the expression is evaluated to be `false`.

Casting operations are usually enclosed in an `if` statement with a condition that uses an `instanceof` operator. For example:

```
if (date instanceof BritishDate5) {
   BritishDate5 bDate = (BritishDate5) date;
   bDate.printLongDate();
}
```

The condition of the `if` statement first verifies whether the object referred to by the variable `date` is a `BritishDate5` object. If the object is a `BritishDate5` object, the expression is evaluated to be `true` and the `if` part of the `if` statement is executed. Then, a variable `bDate` is declared and is assigned the reference that is stored in the variable `date` after casting. The condition of the `if` statement ensures that the object must be a `BritishDate5` object and the casting operation must succeed. Finally, the message `printLongDate` is sent to the `BritishDate5` object via a `BritishDate5` type variable, and the object executes its `printLongDate()` method.

We discussed the `DateHandler1` class with a `saySomethingAboutDate()` method that can accept a `Date1` object. As all its subclass objects can be considered a `Date1` object, the method can accept objects from all its subclasses. That is, the object can be a `Date1` object or an object of its subclasses. If the method has to operate differently for objects of different classes, it can use the `instanceof` operator to determine the actual class of the object and perform the operation accordingly.

The `DateHandler2` class, a modified version of the `DateHandler1` class, is written as shown in Figure 7.137. Its `saySomethingAboutDate()` method uses `instanceof` operators to determine the actual class of the supplied object so that it can behave differently.

```
// Definition of class DateHandler2
public class DateHandler2 {
    // Attributes

    // An array for storing all month names
    private static final String[] MONTH_NAMES = {
        "January", "February", "March", "April",
        "May", "June", "July", "August",
        "September", "October", "November", "December"
    };

    // Methods

    // Show the details of a Date1 object
    public void saySomethingAboutDate(Date1 date) {
        // Check if the object is actually a BritishDate5 object

        if (date instanceof BritishDate5) {
            // It is verified that it is a BritishDate5 object
            System.out.print("The British date (");
            date.print();
            System.out.print(") means ");
            BritishDate5 bDate = (BritishDate5) date;
            bDate.printLongDate();
            System.out.println();
        }
        // Check if the object is actually an AmericanDate5 object

        else if (date instanceof AmericanDate5) {
            // It is verified that it is an AmericanDate5 object
            System.out.print("The American date (");
            date.print();
            System.out.print(") means ");
            AmericanDate5 aDate = (AmericanDate5) date;
            aDate.printLongDate();
            System.out.println();
        }
        // Otherwise
        else {
            System.out.print("The date (");
```

```
            date.print();
            System.out.print(") means ");
            System.out.print(MONTH_NAMES[date.getMonth() - 1]);
            System.out.print(" "  + date.getDay());
            switch (date.getDay()) {
            case 1: System.out.print("st"); break;
            case 2: System.out.print("nd"); break;
            case 3: System.out.print("rd"); break;
            default: System.out.print("th"); break;
            }
            System.out.println(", " + date.getYear() + ".");
        }
    }
}
```

Figure 7.137 DateHandler2.java

The `saySomethingAboutDate()` method of the `DateHandler2` class can accept any `Date1` object. If the supplied object is actually a `BritishDate5` or an `AmericanDate5` object, it can perform differently.

Based on the `TestDateHandler1` class, another class `TestDateHandler2` class is written in Figure 7.138 to test the `DateHandler2` class.

```
// Definition of class TestDateHandler2
public class TestDateHandler2 {

   // Main executive method
   public static void main(String args[]) {
      // Create a DateHandler2 object and is referred by variable
      // handler
      DateHandler2 handler = new DateHandler2();

      // Create a Date1 object
      Date1 date1 = new Date1();
      date1.setDay(1);
      date1.setMonth(2);
      date1.setYear(2001);
      // Call the DateHandler2 object with the Date1 object referred
      // by variable date1
      handler.saySomethingAboutDate(date1);

      // Create a BritishDate5 object
      Date1 date2 = new BritishDate5();
      date2.setDay(3);
      date2.setMonth(4);
      date2.setYear(2002);
      // Call the DateHandler2 object with the BritishDate5 object
      // referred by variable date2
      handler.saySomethingAboutDate(date2);

      // Create an AmericanDate5 object
```

```
        Date1 date3 = new AmericanDate5();
        date3.setDay(5);
        date3.setMonth(6);
        date3.setYear(2003);
        // Call the DateHandler2 object with the AmericanDate5 object
        // referred by variable date3
        handler.saySomethingAboutDate(date3);
    }
}
```

Figure 7.138 TestDateHandler2.java

Compile the classes and execute the `TestDateHandler2` program. The following messages are shown on the screen:

```
The date (20010201) means February 1st, 2001.
The British date (03/04/2002) means 3 April 2002.
The American date (6/5/2003) means June 5, 2003.
```

Compared with the `TestDateHandler1` class, the local variables defined in the `main()` method of the `TestDateHandler2` class are of the type `Date1` and are referring to the three objects of classes `Date1`, `BritishDate5` and `AmericanDate5` respectively. You should now feel comfortable with this, as a `Date1` variable can refer to a `Date1` object and to any object of its subclass.

The `main()` method of the `TestDateHandler2` class creates a `DateHandler2` object. Then, there are three program segments creating a `Date1` object, a `BritishDate5` object and an `AmericanDate5` object respectively, to be supplied to the `saySomethingAboutDate()` method of the `DateHandler2` object.

Based on the output messages, you can see that the `saySomethingAboutDate()` method behaves differently according to the actual class of the supplied object. It uses an `if/else` statement with `instanceof` operators in the method:

```
// Check if the object is actually a BritishDate5 object
if (date instanceof BritishDate5) {
    ......
}
// Check if the object is actually an AmericanDate5 object
else if (date instanceof AmericanDate5) {
    ......
}
// Otherwise
else {
    ......
}
```

In the `main()` method of the `TestDateHandler2` class, the object supplied to the first method call to the `saySomethingAboutDate()` method is a `Date1` object. Therefore, the last block of the `if/else`

statement in the `saySomethingAboutDate()` method is executed as shown in Figure 7.139.

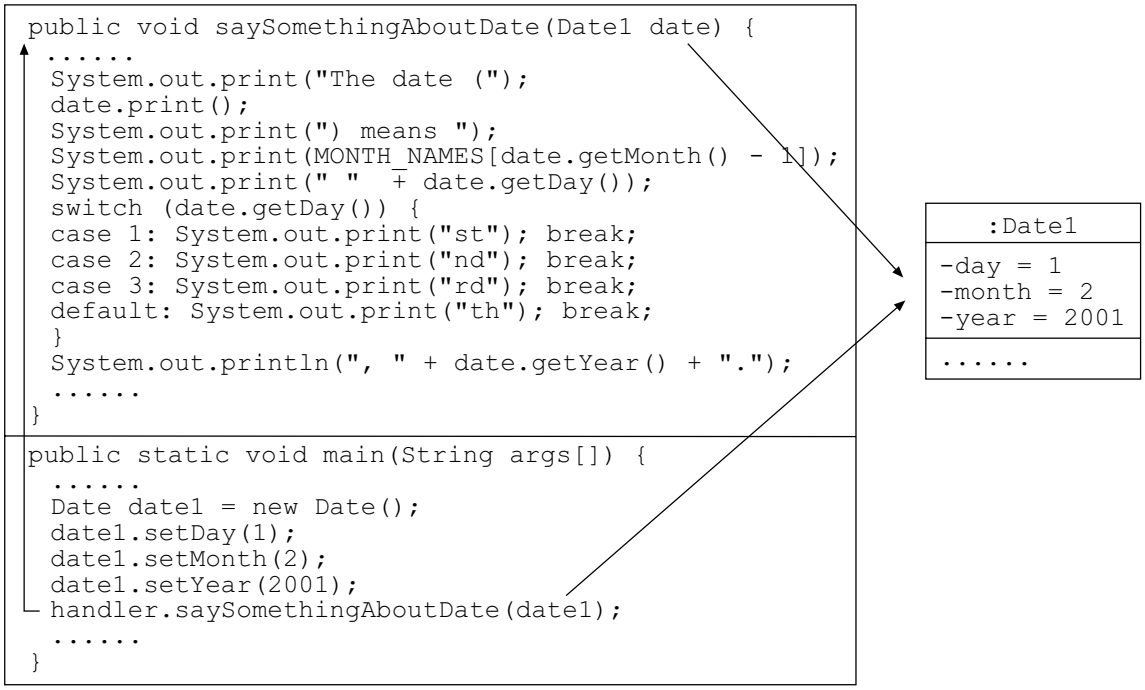


Figure 7.139 The first method call to the `saySomethingAboutDate()` method with a `Date1` object

According to the program segment in the `saySomethingAboutDate()` method and the supplied `Date1` object, the following message is shown on the screen:

```
The date (20010201) means February 1st, 2001.
```

The flow of control is returned to the `main()` method of the `TestDateHandler2` class, and the second program segment is executed to prepare a `BritishDate5` object that is supplied to the second method call of the `saySomethingAboutDate()` method of the `DateHandler2` object.

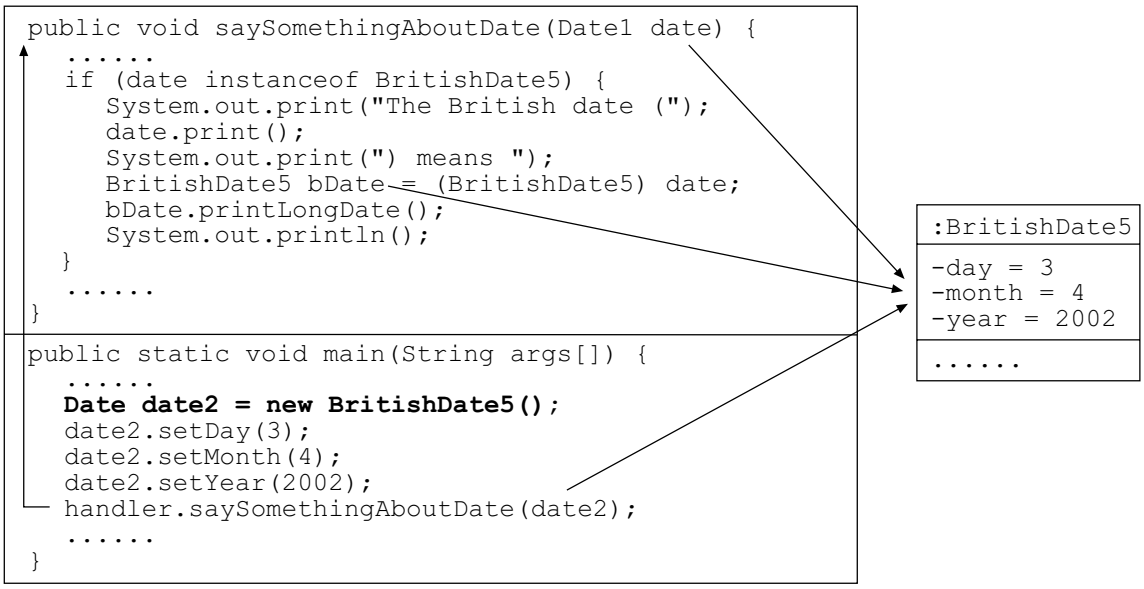


Figure 7.140 The second method call to the `saySomethingAboutDate()` method with a `BritishDate5` object

For the second method call to the `saySomethingAboutDate()` method with the `BritishDate5` object, the parameter `date` is assigned the reference of the `BritishDate5` object. The expression

```
date instanceof BritishDate5
```

is evaluated to be `true`. Then, the corresponding program block is executed and the following message is first shown on the screen:

```
The British date (
```

Afterwards, a message `print` is sent to the `BritishDate5` object that is referred to by the parameter `date`, and the object performs its `print` behaviour according to the method defined in the `BritishDate5` class. The following message is shown:

```
03/04/2002
```

Finally, as the object is determined to be a `BritishDate5` object, the method intends to send a message `printLongDate` to it so that it executes its `printLongDate()` method. As the type of the parameter `date` is `Date1` class that does not define the `printLongDate()` method, it is necessary to assign the reference to a `BritishDate5` variable after casting. Therefore, the statement

```
BritishDate5 bDate = (BritishDate5) date;
```

declares a local variable `bDate` and copies the reference that is stored in the parameter `date` to the `bDate` variable after casting. The condition of the `if/else` statement guarantees that the object must be a `BritishDate5` object and the statement must succeed.

At the end of the program segment, the message `printLongDate` is sent to the `BritishDate5` object via a `BritishDate5` variable

`bDate`, so that the `BritishDate5` object executes its `printLongDate()` method. As a whole, the following message is shown by the second method call to the `saySomethingAboutDate()` method with the `BritishDate5` object:

```
The British date (03/04/2002) means 3 April 2002.
```

Again, the flow of control is returned to the `main()` method of the `TestDateHandler2` class. The last program segment in the `main()` method is executed that prepares an `AmericanDate5` object and calls the `saySomethingAboutDate()` method of the `DateHandler2` object.

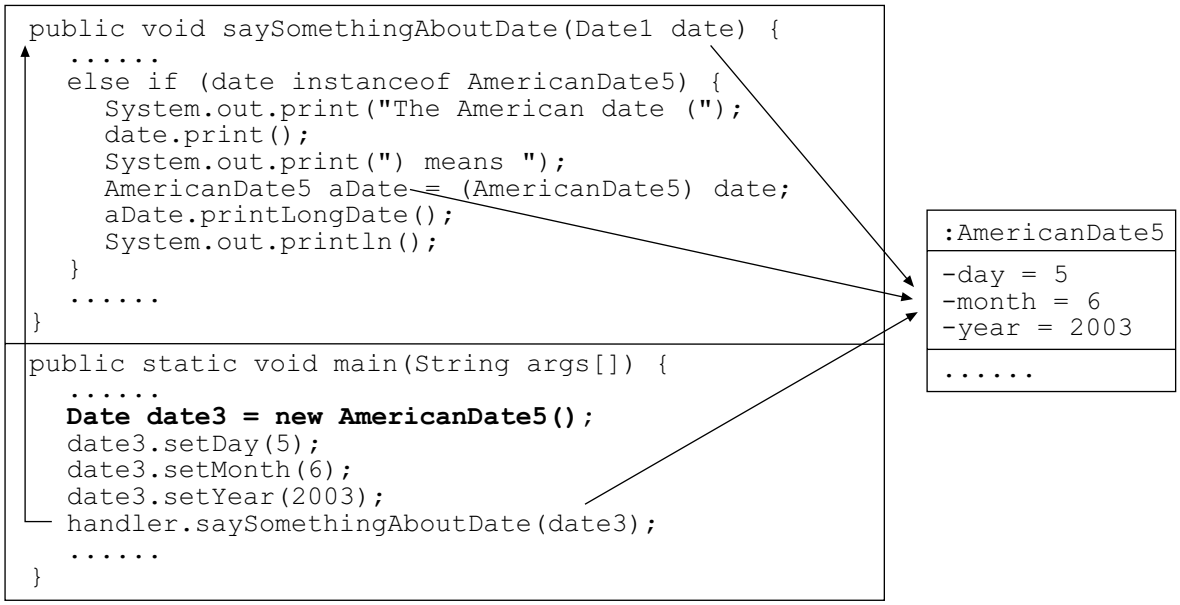


Figure 7.141 The third method call to the `saySomethingAboutDate()` method with an `AmericanDate5` object

The operations in the `saySomethingAboutDate()` method for the third method call with an `AmericanDate5` object are similar to that for the second method call. The object referred by the parameter `date` is an `AmericanDate5` object; hence it performs differently when it receives messages during the execution of the `saySomethingAboutDate()` method. Therefore, the message shown by the third method call to the `saySomethingAboutDate()` method is:

```
The American date (6/5/2003) means June 5, 2003.
```

The `saySomethingAboutDate()` method of the `DateHandler2` class illustrates the use of the `instanceof` operator. Whenever a method has to perform differently according to the actual type of an object, the `instanceof` operator can be used.

Please use the following self-test to practise the use of the `instanceof` keyword.

Self-test 7.14

Based on the `ChineseDate1` class, write a `ChineseDate5` class that defines both `print()` and `printLongDate()` method. Modify the classes `DateHandler2` and `TestDateHandler2`, so that executing the `TestDateHandler2` class shows the following output messages on the screen:

```
The date (20010201) means February 1st, 2001.
The British date (03/04/2002) means 3 April 2002.
The American date (6/5/2003) means June 5, 2003.
The Chinese date (2004.8.7) means 7 August, 2004.
```

In Self-test 7.8, it was not necessary to modify the `DateHandler1` class definition, but it is necessary to modify the `DateHandler2` class definition in Self-test 7.14. Take few minutes to compare the two classes `DateHandler1` and `DateHandler2`. Try to figure out the reason why the `DateHandler2` class needs modification and any implications thereof.

The difference between the `DateHandler1` and `DateHandler2` is that the `saySomethingAboutDate()` method of the `DateHandler2` class uses the `instanceof` operator to determine the actual type of the supplied object and a method specific to the object is called. However, the `saySomethingAboutDate()` of the `DateHandler1` uses the objects as if they are the general type `Date1` and calls on their specific methods.

A method using a multilevel `if/else` statement with `instanceof` operators implies that it accepts an object of a general type but it has to determine the actual type (the specific type of the general type) of the object so that it can perform its behaviour properly. If there is a new specific type of the general type, the method usually needs modification. For example, deriving a new class (`ChineseDate1` class in Self-test 7.8) based on an existing class (`Date1` class), other classes (such as `DateHandler2`) have to be modified as well. It is highly undesirable to modify existing class definitions, because they might be written by other programmers and usually were thoroughly tested. Any modification might not be appropriate and they must be tested again. As a result, if a method uses the `instanceof` operator as in the `DateHandler2` class, there is a risk that it needs modification in the future. You can then investigate the reasons and try to avoid modification.

The core reason why the `saySomethingAboutDate()` method has to use the `instanceof` operator is the `Date1` class does not define the `printLongDate()` method, but its subclasses do. Therefore, if the `Date1` class defines the `printLongDate()` method as well, the `saySomethingAboutDate()` method can be written without the `instanceof` operator. However, there is no universal format for printing a date in long format, and it is therefore unreasonable to define the `printLongDate()` for the `Date1` class. To solve the problem, we

can define the `printLongDate()` method in the `Date1` and mark it `abstract`. Please refer to the section ‘An abstract blueprint — abstract class’ for further details.

Appendix I: calculation of MPF without *polymorphism*

It is possible to perform MFP calculations for a company without using polymorphism. For example, a single class `GeneralStaff` is written to model all staff members as shown in Figure 7.142.

```
// Definition of class GeneralStaff
public class GeneralStaff {
    // Attributes
    private int category;
    private String name;
    private double basicSalary;
    private double salesVolume;
    private double commissionRate;
    private double allowance;

    // Constants for different staff categories
    public static final int CLERK = 1;
    public static final int SALESPERSON = 2;
    public static final int MANAGER = 3;

    // Constructor
    public GeneralStaff(int category, String name,
                        double basicSalary, double salesVolume,
                        double commissionRate, double allowance) {
        this.category = category;
        this.name = name;
        this.basicSalary = basicSalary;
        this.salesVolume = salesVolume;
        this.commissionRate = commissionRate;
        this.allowance = allowance;
    }

    // Get the staff name
    public String getName() {
        return name;
    }

    // Find the salary
    public double findSalary() {
        switch (category) {
        case CLERK:
            return basicSalary;
        case SALESPERSON:
            return basicSalary + salesVolume * commissionRate;
        case MANAGER:
            return basicSalary + allowance;
        default:
            return 0.0;
        }
    }
```

```
    // Find the net salary after MPF contribution
    public double findNetSalary() {
        double salary = findSalary();
        if (salary < 5000.0) {
            return salary;
        }
        else if (salary >= 20000.0) {
            return salary - 1000.0;
        }
        else {
            return salary * 0.95;
        }
    }

    // Find the MPF contribution by the company
    public double findMPFByCompany() {
        return (findSalary()>20000.0)? 1000.0 : findSalary()* 0.05;
    }

    // Find the MPF contribution by the staff
    public double findMPFByStaff() {
        double salary = findSalary();
        if (salary < 5000.0) {
            return 0.0;
        }
        else if (salary >= 20000.0) {
            return 1000.0;
        }
        else {
            return salary * 0.05;
        }
    }
}
```

Figure 7.142 GeneralStaff.java

Notice no `break` statements are needed in the `findSalary()`method, since the `return` statement causes the control to be returned to the caller. The equivalent payroll calculation and MPF contributions are performed in the `PayrollCalculator2` as shown in Figure 7.143.

```
// Definition of class PayrollCalculator2
public class PayrollCalculator2 {
    // Attribute
    private GeneralStaff[] staffList; // The staff list to be
processed

    // Constructor
    public PayrollCalculator2(GeneralStaff[] staffList) {
        this.staffList = staffList;
    }
    public void showReport() {
        // Declare and initialize running totals
        double totalSalary = 0.0;
```

```
        double totalMPFByStaff = 0.0;
        double totalMPFByCompany = 0.0;

        // Show the listing title
        System.out.println("Staff\tRaw\tNet\tMPF by\tMPF by");
        System.out.println("Name\tSalary\tSalary\tStaff\tCompany");
        System.out.println("------ ------- ------ ------ ------");

        // Iterate each staff to show his/her details
        for (int i=0; i < staffList.length; i++) {
            // Get the staff payroll details
            double salary = staffList[i].findSalary();
            double netSalary = staffList[i].findNetSalary();
            double mpfByStaff = staffList[i].findMPFByStaff();
            double mpfByCompany = staffList[i].findMPFByCompany();

            // Show the details
            System.out.println(
                staffList[i].getName() +
                "\t" + salary +
                "\t" + netSalary +
                "\t" + mpfByStaff +
                "\t" + mpfByCompany);

            // Update the running totals
            totalSalary += netSalary;
            totalMPFByStaff += mpfByStaff;
            totalMPFByCompany += mpfByCompany;
        }

        // Show the grand totals
        System.out.println("------- ------ ------ ------ -------");
        System.out.println(
            "Total" +
            "\t\t" + totalSalary +
            "\t" + totalMPFByStaff +
            "\t" + totalMPFByCompany);
    }
}
```

Figure 7.143 PayrollCalculator2.java

To test the `PayrollCalculator2` class, a driver program `TestPayrollCalculator2` is written in Figure 7.144.

```
// Definition of class TestPayrollCalculator2 {
public class TestPayrollCalculator2 {

    // Main exeuctive method
    public static void main(String args[]) {
        // Create an array of staff in the company
        GeneralStaff[] staff = {
            new GeneralStaff(GeneralStaff.CLERK,
                "Mary", 4000.0, 0.0, 0.0, 0.0),
            new GeneralStaff(GeneralStaff.CLERK,
                "Peter", 6000.0, 0.0, 0.0, 0.0),
            new GeneralStaff(GeneralStaff.SALESPERSON,
                "Joe", 4000.0, 100000.0, 0.05, 0.0),
            new GeneralStaff(GeneralStaff.SALESPERSON,
                "Amy", 5000.0, 200000.0, 0.08, 0.0),
            new GeneralStaff(GeneralStaff.MANAGER,
                "John", 20000.0, 0.0, 0.0, 10000.0)
        };

        // Create a PayrollCalculator2 object by supplying an array
        // of Staff objects
        PayrollCalculator2 calculator = new PayrollCalculator2(staff);

        // Show the payroll report
        calculator.showReport();
    }
}
```

Figure 7.144 TestPayrollCalculator2.java

Comparing the `main()` methods of the `TestPayrollCalculator1` and `TestPayrollCalculator2`, the ways of creating the staff lists are different. The `TestPayrollCalculator2` class creates the staff list with the following program segment:

```
// Create an array of staff in the company
GeneralStaff[] staff = {
    new GeneralStaff(GeneralStaff.CLERK,
        "Mary", 4000.0, 0.0, 0.0, 0.0),
    new GeneralStaff(GeneralStaff.CLERK,
        "Peter", 6000.0, 0.0, 0.0, 0.0),
    new GeneralStaff(GeneralStaff.SALESPERSON,
        "Joe", 4000.0, 100000.0, 0.05, 0.0),
    new GeneralStaff(GeneralStaff.SALESPERSON,
        "Amy", 5000.0, 200000.0, 0.08, 0.0),
    new GeneralStaff(GeneralStaff.MANAGER,
        "John", 20000.0, 0.0, 0.0, 10000.0)
};
```

As the `GeneralStaff` class defines all possible attributes for all staff categories, it is necessary to provide sufficient parameter values to the constructors. Therefore, you can see that it is necessary to supply dummy values to the constructor if that staff category does not use those attributes. Furthermore, an important observation is that it is necessary to provide a value to the constructor to identify the staff category. The

programmer who uses the `GeneralStaff` class must thoroughly understand the relationship between the category value and the other values supplied to the constructor that can be error-prone. Furthermore, it is not a good idea for an object to possess unnecessary attributes.

Another problem with the implementation is the extensibility of the program. If there is a new staff category, say hourly waged staff, new attributes and a constant have to be added to the `GeneralStaff` class; the constructor parameter list and the `switch/case` statement in the `findSalary()` method need modification. As we have said, modifying an existing tested class definition needs retesting. It is error-prone and the software reliability is not guaranteed.

Please use the following self-test to test adding a new staff category to the payroll and MPF calculation without polymorphism. Compared with the answer for Self-test 7.10, you will appreciate the beauty of polymorphism and the use of abstract superclasses with concrete subclasses.

Self-test 7.15

1 Modify the `GeneralStaff` to incorporate a new temporary staff category and the `TestPayrollCalculator2` class as mentioned in Self-test 7.10.

2 Comment on the necessary modifications for the implementations with polymorphism (abstract superclass with concrete subclasses) and without polymorphism.

Suggested answers to self-test questions

Self-test 7.1

1 The types for attributes year and month can be `int`, and the types for staff number and salary can be `String` and `double` respectively.

2 The ranges for attributes year and month can be determined to be `1900` to `9999` and `1` to `12` respectively. For the attribute staff number, its value must not be `null`. As salary is always non-negative, its value must be greater than or equal to `0`.

3 **PayrollRecord.java**

```
// Definition of class PayrollRecord
public class PayrollRecord {
    private int year;          // the payroll year
    private int month;         // the payroll month
    private String staffNumber;// the staff number
    private double salary;     // the salary for the month

    // Getter methods

    // Get the payroll year
    public int getYear() {
        return year;
    }

    // Get the payroll month
    public int getMonth() {
        return month;
    }

    // Get the staff number
    public String getStaffNumber() {
        return staffNumber;
    }

    // Get the salary
    public double getSalary() {
        return salary;
    }

    // Setter methods

    // Set the payroll year
    public void setYear(int theYear) {
        if (theYear >= 1900 && theYear <= 9999) {
            year = theYear;
        }
        else {
            System.out.println("Invalid year.");
        }
```

```
    }

    // Set the payroll month
    public void setMonth(int theMonth) {
        if (theMonth >= 1 && theMonth <= 12) {
            month = theMonth;
        }
        else {
            System.out.println("Invalid month");
        }
    }

    // Set the staff number
    public void setStaffNumber(String theStaffNumber) {
        if (theStaffNumber != null) {
            staffNumber = theStaffNumber;
        }
        else {
            System.out.println("Invalid staff number");
        }
    }

    // Set the salary for the month
    public void setSalary(double theSalary) {
        if (theSalary >= 0.0) {
            salary = theSalary;
        }
        else {
            System.out.println("Invalid salary");
        }
    }
}
```

Self-test 7.2

1 **IntegerStack5.java**

```
// Definition of the class IntegerStack (version 5)
public class IntegerStack5 {
    // Attributes
    // The storage for the numbers in the stack
    private int[] storage;
    // The attribute for the subscript of the element that can store
    // the newly added number
    private int top;

    // Behaviours
    // The constructor for the class
    public IntegerStack5(int initialCapacity) {
      storage = new int[initialCapacity];
   }

    // The behaviour to push a new number
    public void push(int number) {
        // Show debug message
        System.out.println("DEBUG: Push " + number);
```

```
        // Verify whether all array elements have been used
        if (top == storage.length) {
            // Create a new array with larger size
            int[] newArray = new int[storage.length + 1];
            // Use a loop to copy the element contents from the
            // original array to the new one.
            for (int i=0; i < storage.length; i++) {
                newArray[i] = storage[i];
            }
            // Update the attribute storage, so that this IntegerStack
            // object is referring to the new array object
            storage = newArray;
        }
        // Store the number and increase the subscript for
        // storing the next number afterwards
        storage[top++] = number;
    }

    // The behaviour to pop the last number
    public int pop() {
        // Show debug message
        System.out.println("DEBUG: Pop");

        // A local variable result is declared to store the value
        // to be returned
        int result = -1;

        // Verify whether the stack is empty
        if (top > 0) {
            // Decrease the subscript for the last number and
            // Store the last number to local variable result afterwards
            result = storage[--top];
        }
        else {
            // Show message to prompt the user
            System.out.println("The stack is empty");
        }
        return result;
    }
}
```

2 **Item.java**

```
// Definition of class Item
public class Item {
    // Attributes
    private String name;
    private double price;
    private int quantity;

    // Behaviours

    // The constructor of the class
    public Item(String theName, double thePrice, int theQuantity) {
      name = theName;
       price = thePrice;
       quantity = theQuantity;
   }

    // Set the attribute name
    public void setName(String theName) {
        name = theName;
    }

    // Set the attribute price
    public void setPrice(double thePrice) {
        price = thePrice;
    }

    // Set the attribute quantity
    public void setQuantity(int theQuantity) {
        quantity = theQuantity;
    }

    // Get the subtotal of this item
    public double findTotal() {
        return price * quantity;
    }
}
```

PurchaseOrder.java

```
// Definition of class PurchaseOrder
public class PurchaseOrder {
    // Attributes
    private Item[] items;
    private int itemCount;

    // Behaviours
    // The constructor
    public PurchaseOrder(int itemTotal) {
                    items = new Item[itemTotal];
                    }

    // To create and add a new item to the array object
    public void addItem(String name, double price, int quantity) {
        // Create a new Item object
        Item newItem = new Item(name, price, quantity);

        // Add the Item object to the array object
        items[itemCount++] = newItem;
    }

    // Get the total of all items
    public double findTotal() {
        double sum = 0.0;
        for (int i=0; i < itemCount; i++) {
            sum += items[i].findTotal();
        }
        return sum;
    }
}
```

Cashier.java

```
// Definition of class Cashier
public class Cashier {

    // Main executive method
    public static void main(String args[]) {
        // Create a new PurchaseOrder object to be referred by
        // the local variable order
        PurchaseOrder order = new PurchaseOrder(3);

        // Add the items to the purchase order
        order.addItem("Coke", 2.5, 12);
        order.addItem("Milk", 6.0, 1);
        order.addItem("Egg", 0.5, 18);

        // Display the total price
        System.out.println("Total price = " + order.findTotal());
    }
}
```

Self-test 7.3

1 a A car ‘has an’ Engine.

b A Personal Digital Assistant ‘is a’ computer.

c A Cat ‘is a’ Pet.
A Dog ‘is a’ Pet.

d A Shirt ‘is a’ Clothing.
A Skirt ‘is a’ Clothing.

2 **Account.java**

```
// Definition of class Account
public class Account {
    // Attributes
    private String ownerName;// the name of the account owner
    private double balance;  // the balance of the account

    // Getter methods

    // Get the name of the account owner
    public String getOwnerName() {
        return ownerName;
    }

    // Get the account balance
    public double getBalance() {
        return balance;
    }

    // Setter methods

    // Set the name of the account owner
    public void setOwnerName(String theOwnerName) {
        ownerName = theOwnerName;
    }

    // Set the account balance
    public void setBalance(double theBalance) {
        balance = theBalance;
    }
}
```

ODAccount.java

```
// Class definition of class ODAccount
public class ODAccount extends Account {
    // Attribute
    private double overDrawnLimit;// The overdrawn limit

    // Getter method

    // Get the overdrawn limit
    public double getOverDrawnLimit() {
        return overDrawnLimit;
    }

    // Setter method

    // Set the overdrawn limit
    public void setOverDrawnLimit(double theOverDrawnLimit)
{
        overDrawnLimit = theOverDrawnLimit;
    }
}
```

Self-test 7.4

1 **TicketCounter.java**

```
// The class definition of a ticket counter
class TicketCounter {
    private int reading; // The ticket counter reading

    // To get the ticket counter reading
    public int getReading() {
        return reading;
    }

    // To increase the reading by one
    public void increase() {
        reading = reading + 1;
    }

    // To increase the reading specified by the parameter, amount
    public void increase(int amount) {
        reading = reading + amount;
    }

    // To set a reading to the ticket counter reading
    public void setReading(int newReading) {
        reading = newReading;
    }
}
```

2 **TestCounter.java**

```
// Definition of class TestCounter
public class TestCounter {

    // Main executive method
    public static void main(String args[]) {
        TicketCounter counter1, counter2;
        counter1 = new TicketCounter();
        counter2 = counter1;
        counter1.setReading(10);
        counter1.increase();
        counter2.increase(2);
        System.out.println("The Reading of counter1.reading is " +
            counter1.getReading());
    }
}
```

Self-test 7.5

Time3.java

```
// Definition of class Time3
public class Time3 {
   // Attributes
    private int hour;      // The hour of time
    private int minute;    // The minute of time
    private int second;    // the second of time

    // The final class variables
    private static final int DEFAULT_HOUR = 0;
    private static final int DEFAULT_MINUTE = 0;
    private static final int DEFAULT_SECOND = 0;

    // Constructors
    public Time3() {
        this(DEFAULT_HOUR, DEFAULT_MINUTE, DEFAULT_SECOND);
    }

    public Time3(int theHour, int theMinute, int theSecond) {
        hour = theHour;
        minute = theMinute;
        second = theSecond;
    }

    // Get the value of attribute hour
    public int getHour() {
       return hour;
   }

   // Get the value of attribute minute
   public int getMinute() {
      return minute;
   }

   // Get the value of attribute second
```

```
    public int getSecond() {
      return second;
    }

    // Set the value of attribute hour
    public void setHour(int theHour) {
      if (0 <= theHour && theHour < 24) {
          hour = theHour;
        }
        else {
          System.out.println(
           "Invalid hour. Object attribute kept unchanged.");
      }
    }

    // Set the value of attribute minute
    public void setMinute(int theMinute) {
      if (0 <= theMinute && theMinute < 60) {
        minute = theMinute;
      }
      else {
        System.out.println(
          "Invalid minute. Object attribute kept unchanged.");
      }
    }

    // Set the value of attribute second
    public void setSecond(int theSecond) {
      if (0 <= theSecond && theSecond < 60) {
        second = theSecond;
      }
      else {
        System.out.println(
          "Invalid second. Object attribute kept unchanged.");
      }
    }
}
```

Self-test 7.6

1 **Item1.java**

```
// Definition of class Item1
public class Item1 {
    // Default values
    private static final String DEFAULT_NAME = "Untitled";
    private static final double DEFAULT_PRICE = 0.0;
    private static final int DEFAULT_QUANTITY = 1;

    // Attributes
    private String name;
    private double price;
    private int quantity;

    // Behaviours

    // The constructors of the class
    public Item1() {
        // Call the constructor with an extra default name
        this(DEFAULT_NAME);
    }

    public Item1(String name) {
        // Call the constructor with an extra default price
        this(name, DEFAULT_PRICE);
    }

    public Item1(String name, double price) {
        // Call the constructor with an extra default quantity
        this(name, price, DEFAULT_QUANTITY);
    }

    public Item1(String name, double price, int quantity) {
        // Set the attributes properly
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

    // Set the attribute name
    public void setName(String name) {
        this.name = name;
    }

    // Set the attribute price
    public void setPrice(double price) {
        this.price = price;
    }

    // Set the attribute quantity
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    // Get the subtotal of this item
    public double findTotal() {
        return price * quantity;
    }
}
```

2 With respect to the constructor,

```
public TicketCounter4(int raeding) {
    this.reading = reading % LIMIT;
}
```

the parameter name of the method is `raeding` instead of `reading`. Therefore, the expression `this.reading` and `reading` in the statement

```
this.reading = reading % LIMIT;
```

refers to the object attribute `reading`, and it is therefore not a compilation error. When the constructor is executed, the value of the attribute `reading` is implicitly initialized to be `0`. The expression `reading % LIMIT` is evaluated to be `0` and the result is assigned to the attribute `reading`. As a result, the value of the attribute `reading` is actually unchanged. Therefore, the value of the attribute `reading` of the created `TicketCounter4` object is always zero no matter what value is supplied to the constructor.

Such mistakes cannot be determined by the compiler software and are quite difficult to determine (find). The following constructor

```
public TicketCounter4(int theReading) {
    reading = theReading % LIMIT;
}
```

is equivalent to the following one:

```
public TicketCounter4(int reading) {
    this.reading = reading % LIMIT;
}
```

The difference is that if the parameter name of the former was mistyped, the compiler software prompts you with a compilation error, whereas the latter would not. It is then up to you to choose your own way of implementation.

Self-test 7.7

ChineseDate1.java

```
// Definition of class ChineseDate1
public class ChineseDate1 extends Date1 {

    // Override the print() method defined in the Date1 class
    // and show the date in dd/mm/yyyy format
    public void print() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in yyyy.m.d format
        System.out.print(year + "." + month + "." + day);
    }
}
```

Self-test 7.8

ChineseDate1.java

```
// Definition of class ChineseDate1
public class ChineseDate1 extends Date1 {

    // Override the print() method defined in the Date1 class
    // and show the date in yyyy.m.d format
    public void print() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in yyyy.m.d format
        System.out.print(year + "." + month + "." + day);
    }
}
```

TestDateHandler1.java

```
// Definition of class TestDateHandler1
public class TestDateHandler1 {

   // Main executive method
   public static void main(String args[]) {
      // Create a DateHandler1 object and is referred by variable
      // handler
      DateHandler1 handler = new DateHandler1();

      // Create a Date1 object
      Date1 date1 = new Date1();
      date1.setDay(1);
      date1.setMonth(2);
      date1.setYear(2001);
      // Call the DateHandler1 object with the Date1 object referred
      // by variable date1
      handler.saySomethingAboutDate(date1);

      // Create a BritishDate1 object
      BritishDate1 date2 = new BritishDate1();
      date2.setDay(3);
      date2.setMonth(4);
      date2.setYear(2002);
      // Call the DateHandler1 object with the BritishDate1 object
      // referred by variable date2
       handler.saySomethingAboutDate(date2);

      // Create an AmericanDate1 object
      AmericanDate1 date3 = new AmericanDate1();
      date3.setDay(5);
      date3.setMonth(6);
      date3.setYear(2003);
      // Call the DateHandler1 object with the AmericanDate1 object
      // referred by variable date3
       handler.saySomethingAboutDate(date3);
```

```
     // Create an ChineseDate object
     ChineseDate date4 = new ChineseDate();
     date4.setDay(7);
     date4.setMonth(8);
     date4.setYear(2004);
     // Call the DateHandler1 object with the ChineseDate object
     // referred by variable date4
       handler.saySomethingAboutDate(date4);
   }
}
```

Self-test 7.9

AmericanDate2.java

```
// Definition of class AmericanDate2
public class AmericanDate2 extends Date2 {

    // Constructor
    public AmericanDate2(int day, int month, int year) {
        super(day, month, year);
    }

    // Override the print() method defined in the Date1 class
    // and show the date in m/d/yyyy format
    public void print() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in m/d/yyyy format
        System.out.print(month + "/" + day + "/" + year);
    }
}
```

Self-test 7.10

TempStaff.java

```
// Definition of class Clerk
public class TempStaff extends Staff {
    // Attributes
    private double hourlyWage;
    private double workingHours;

    // Constructor
    public TempStaff(String name, double hourlyWage,
                     double workingHours) {
        super(name, 0.0);    // No basic salary
        this.hourlyWage = hourlyWage;
        this.workingHours = workingHours;
    }

    // Method

    // Find the raw salary
```

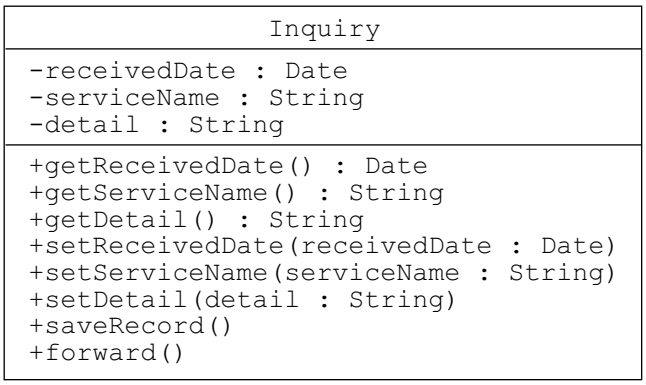
```
    public double findSalary() {
        return hourlyWage * workingHours;
    }
}
```

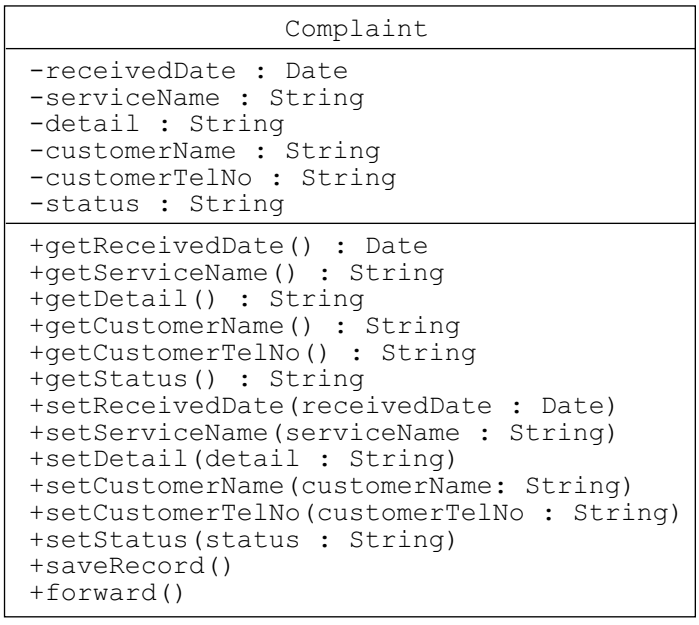
TestPayrollCalculator1.java

```
// Definition of class TestPayrollCalculator1 {
public class TestPayrollCalculator1 {

    // Main exeuctive method
    public static void main(String args[]) {
        // Create an array of staff in the company
        Staff[] staff = {
            new Clerk("Mary", 4000.0),
            new Clerk("Peter", 6000.0),
            new SalesPerson("Joe", 4000.0, 100000.0, 0.05),
            new SalesPerson("Amy", 5000.0, 200000.0, 0.08),
            new Manager("John", 20000.0, 10000.0),
            new TempStaff("Benny", 50.0, 160)
        };

        // Create a PayrollCalculator1 object by supplying an array
        // of Staff objects
        PayrollCalculator1 calculator = new PayrollCalculator1(staff);

        // Show the payroll report
        calculator.showReport();
    }
}
```

Self-test 7.11

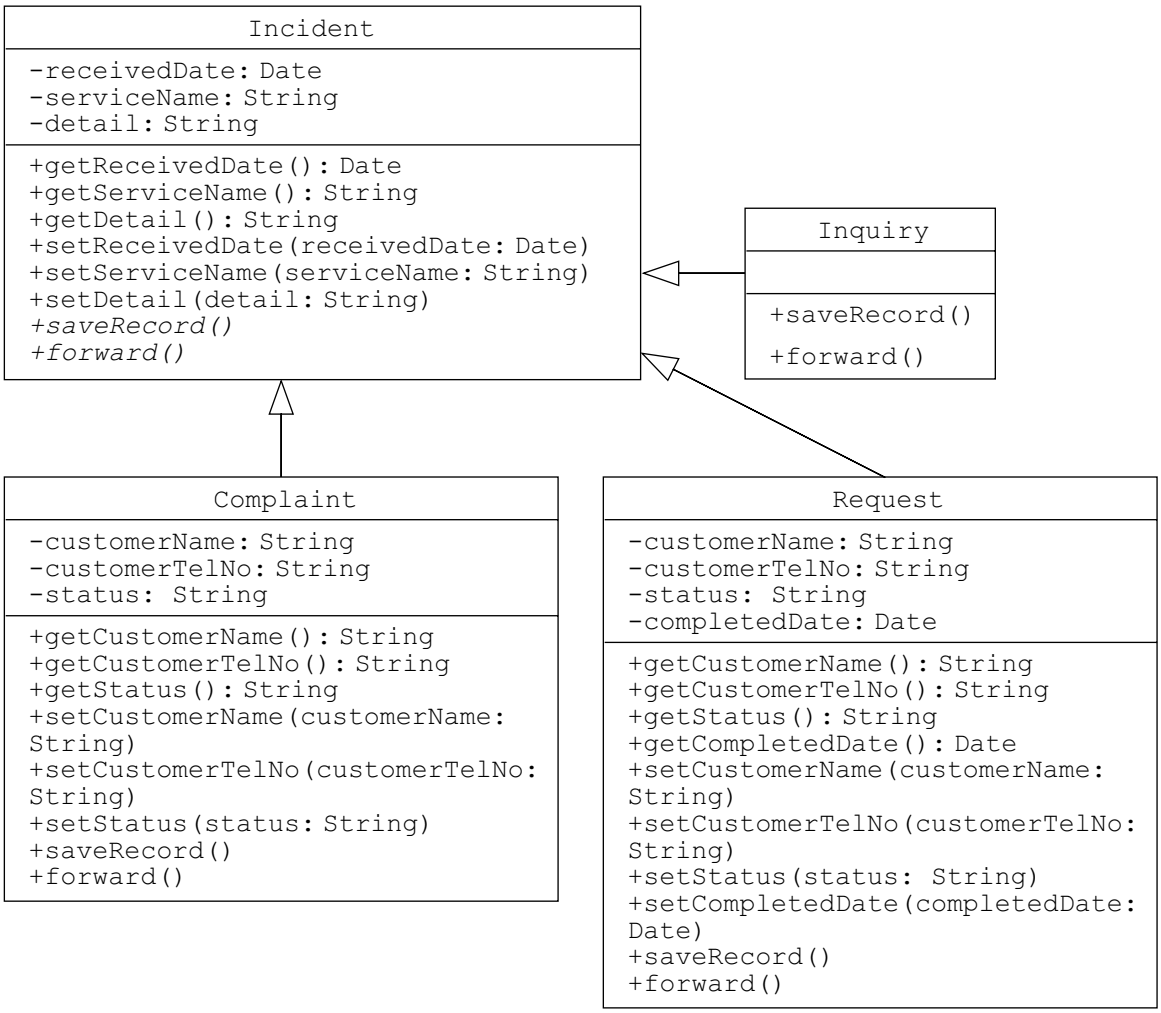
1 There are three types of phone call received by a staff member of the customer service department. They are, inquiry, request and complaint. Therefore, we need three classes, `Inquiry`, `Request` and `Complaint`, to models the calls. On further investigation, the following class designs are obtained.





2 Analysing the classes, it is found that there are no ‘is a’ relationships among them. However, there are similarities among the classes. Therefore, it is possible to derive a general class to be the superclass

of the three classes. It is obvious that inquiry, request and complaint are an incident. As a result, we can derive an `Incident` class to be a superclass of the three classes.



The method `saveRecord()` and `forward()` in the `Incident` class are declared `abstract` because different types of user interaction may be kept in different storage, and different types of incident are handled differently — and probably forwarded to different departments. As a result, it is preferable to define these two methods `abstract`.

Different software developers may come up with different class designs. It greatly depends on the analysis and the interpretation of the requirements.

Self-test 7.12

1 It is logical that the results of `counter1.equals(counter2)` should always equal `counter2.equals(counter1)`. However, there are times when the results can be different. You should notice that the core difference between the two statements is that the message `equals` is sent to different objects, but the two variables are not necessarily referring to objects of the same class. Therefore, the possible scenarios for the return values of the two statements are:

a The two variables are referring to objects of different classes, provided that the classes are either `TicketCounter6` class or its subclasses. Therefore, if a subclass defines an overriding `equals()` method, the two `equals()` methods called in the two statements are not the same one, and the results can be different.

For example, a new class `MyTicketCounter6` is defined as the subclass of `TicketCounter6`, such as:

```
public class MyTicketCounter6 extends TicketCounter6 {
  public boolean equals(Object obj) {
     return false;
  }
}
```

Furthermore, the variables `counter1` and `counter2` are declared and initialized as:

```
TicketCounter6 counter1 = new TicketCounter6();
TicketCounter6 counter2 = new MyTicketCounter6();
```

Then, the return values of `counter1.equals(counter2)` and `counter2.equals(counter1)` will be different, because the return value of the latter one is always `false`.

b If the value of either one of the variables `counter1` or `counter2` is `null`, either `counter1.equals(counter2)` or `counter2.equals(counter1)` will cause a runtime error, because a message `equals` is sent to a non-existing object to execute its `equals()` method.

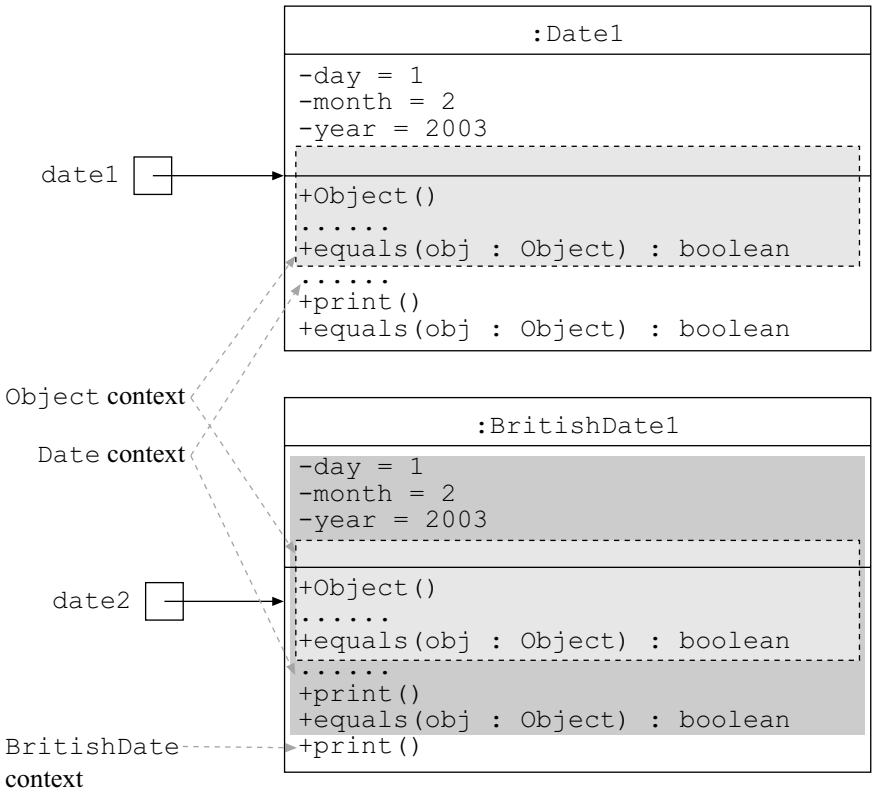
2 **Date1.java**

```
// definition of the class Date1
public class Date1 {
    // Attributes
    private int day;        // the day
    private int month;      // the month
    private int year;       // the year

    // Getter methods

    // get the day
    public int getDay() {
        return day;
    }

    // get the month
    public int getMonth() {
        return month;
    }

    // get the year
    public int getYear() {
```

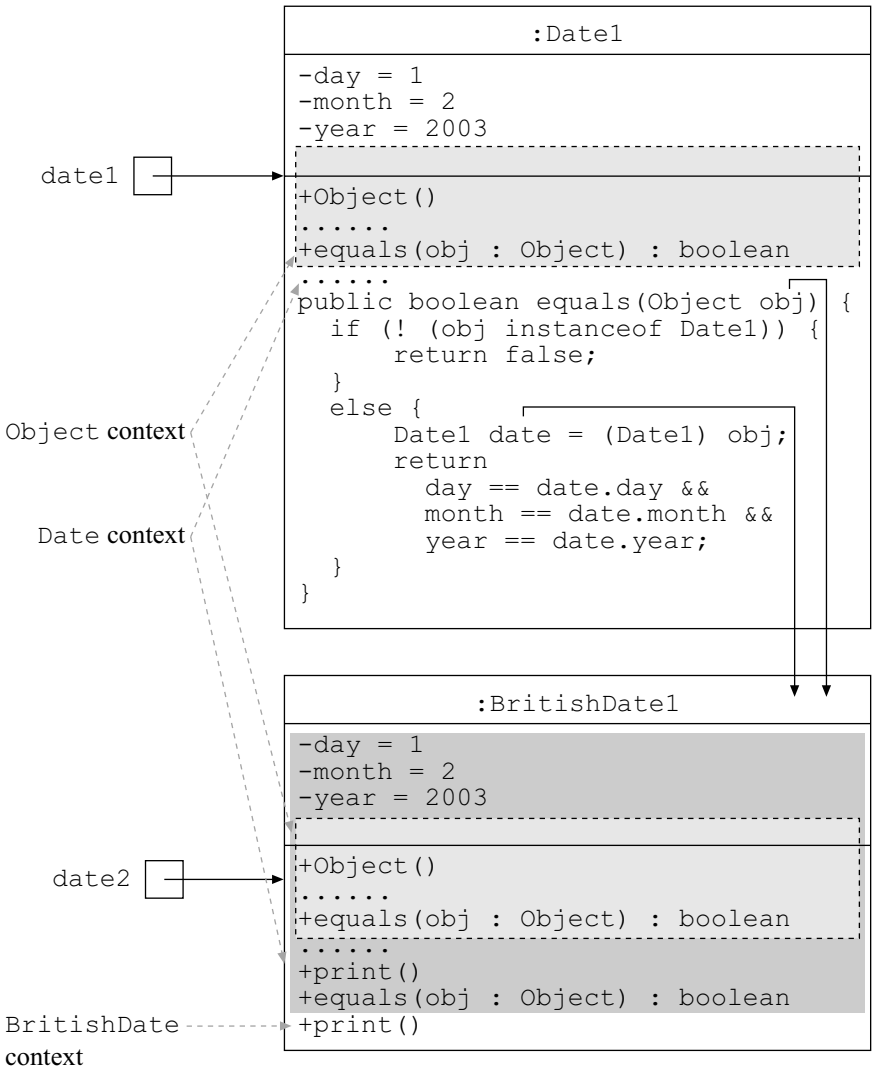
```
        return year;
    }

    // Setter methods

    // set the day
    public void setDay(int day) {
        this.day = day;
    }

    // set the month
    public void setMonth(int month) {
        this.month = month;
    }

    // set the year
    public void setYear(int year) {
        this.year = year;
    }

    // Display the date in yyyymmdd format on the screen
    public void print() {
        System.out.print(year);
        System.out.print((month < 10) ? "0" + month : "" + month);
        System.out.print((day < 10) ? "0" + day : "" + day);
    }

    // Check whether the object equals
    public boolean equals(Object obj) {
        // Check if the supplied object is a Date1 object
        if (! (obj instanceof Date1)) {
            // If it is not a Date1 object, return false
            return false;
        }
        else {
            // If it is a Date1 object, cast the reference
            Date1 date = (Date1) obj;
            // Return the comparison result
            return
                day == date.day &&
                month == date.month &&
                year == date.year;
        }
    }
}
```

3 The `Date1` class defines the `equals()` method, but its subclass `BritishDate1` defines no `equals()` method. Therefore, the `equals()` method to be executed by both `Date1` object and `BritishDate1` object is the same one defined in the `Date2` class.

After the `Date1` object and `BritishDate1` object are created, the scenario is shown as:



While the statement `date1.equals(date2)` is executing, the reference of the `BritishDate1` object is supplied to the `equals()` method for the `Date1` object to execute the `equals()` method. That is:



Executing the `equals()` method defined in the `Date1` class, the condition

```
obj instanceof Date1
```

is evaluated first. The condition is evaluated to be `true` because the parameter `obj` is referring to a `BritishDate1` that can be treated as a `Date1` object. The `!` operator negates the result from `true` to `false` and the `else` part of the `if/else` statement is executed.

Then, a local variable `date` is declared, and the reference `obj` is casted from `Object` class to `Date1` class. The problem here is, is it possible to cast a `BritishDate1` reference to a `Date1` reference? Like the `instanceof` operator, as the `BritishDate1` object can be treated as a `Date1` object and the `BritishDate1` object possesses all methods and attributes defined in the `Date1` class, the JVM allows the casting operation, and the local variable `date` is then referring to the `BritishDate1` object as if it were a `Date1` object.

The method execution continues, and the subsequent comparisons verify whether the attribute values of the `Date1` object (actually a

`BritishDate1` object in this case) equal those of the `Date1` object (referred by the `date1` variable in the `main()` method) itself. The overall result is `true` and is returned. Therefore, the statement

```
date1.equals(date2)
```

returns `true` and is shown on the screen.

For the second statement

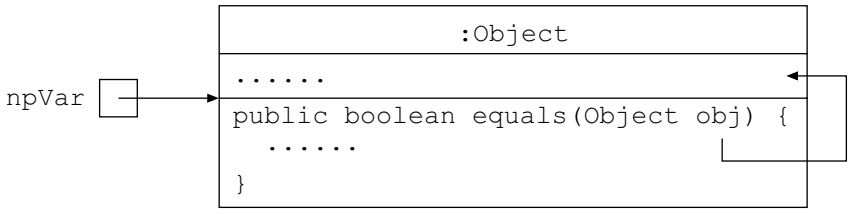
```
date2.equals(date1)
```

the `equals()` method is the same one that is defined in the `Date1` class. The scenario is obvious in this instance, because the object supplied to the `equals()` method is a strictly `Date1` object. Therefore, it is simpler than the previous one, and you can trace the execution for yourself. The return value of the second statement is also `true`.

4 In Question 3, you can see that the `equals()` method defined in the `Date1` class is applicable to the subclass `BritishDate1` and the `equals()` method can verify the equality of objects of `Date1` class and its subclasses. Therefore, provided that the subclass defined no new attribute and the definition of equality remains unchanged, it is not necessary to define overriding `equals()` methods in the subclasses.

5 If the variable `npVar` is non-primitive, it can refer to an object that must be a subclass of `Object` class and possesses an `equals()` method, either defined by it or inherited from its superclass. Furthermore, the `equals()` method can accept a reference to `Object` class objects, which means that it can accept objects of its subclasses, that is, all classes in the Java programming language. Therefore, it is possible to supply the reference stored in the `npVar` variable. As a result, the compiler software will always accept the statement `npVar.equals(npVar)`.

Provided that the content of the variable `npVar` is not `null`, the object referred by the `npVar` variable is supplied with a reference stored in variable `npVar`. That is:



If the `equals()` method is properly defined, the `equals()` method is actually comparing the same object that is referred to by the variable `npVar` and it should always return `true`.

However, if the content of the variable `npVar` is `null`, it will cause a runtime error, because the statement intends to send an `equals` message to a non-existing object.

Self-test 7.13

1 In most cases, `System.out.println(obj)` and `System.out.println(obj.toString())` are practically equivalent. However, there are two differences:

a If the type of the variable `obj` is neither `char[]` nor `String`, the `System.out.println()` method to be executed with respect to `System.out.println(obj)` is the one that accepts an `Object` class object. However, if the statement is modified to be `System.out.println(obj.toString())`, the `System.out.println()` method to be called is the one that accepts a `String` object as supplementary data.

b If the content of the variable `obj` is `null`, the statement `System.out.println(obj)` will display `null` on the screen. By contrast, the statement `System.out.println(obj.toString())` will cause a runtime error, because it is not possible to send a `toString` message to a non-existing object (the reference stored in the variable `obj` is `null`).

2 **TicketCounter6.java**

```
// Definition of class TicketCounter6
public class TicketCounter6 {
   // Attribute
   public static final int LIMIT = 1000;
   private int reading; // The reading of the counter

   // The constructors
   public TicketCounter6() {
       reading = 0;
   }

   public TicketCounter6(int reading) {
      this.reading = reading % LIMIT;
   }

   // Increase the reading by one
   public void increase() {
      reading = (++reading) % LIMIT;
   }

   // Increase the reading by the specified amount
   public void increase(int amount) {
      reading = (reading + amount) % LIMIT;
   }

   // Retrieve the value of the attribute reading
```

```
        public int getReading() {
           return reading;
        }

        // Check whether two objects are equal
        public boolean equals(Object obj) {
            if (! (obj instanceof TicketCounter6)) {
                return false;
            }
            TicketCounter6 counter = (TicketCounter6) obj;
            return reading == counter.reading;
        }

        // Return a textual representation of the object
        public String toString() {
            return "reading=" + reading;
        }
    }
```

TestToString.java

```
// Definition of class TestToString
public class TestToString {

    // Main executive method
    public static void main(String args[]) {
        TicketCounter6 counter = new TicketCounter6(100);
        System.out.println(counter);
    }
}
```

Self-test 7.14

ChineseDate5.java

```
// Definition of class ChineseDate5
public class ChineseDate5 extends Date1 {
    // Attributes

    // An array for storing all month names
    private static final String[] MONTH_NAMES = {
        "January", "February", "March", "April",
        "May", "June", "July", "August",
        "September", "October", "November", "December"
    };

    // Methods

    // Print the date in short format
    public void print() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in m/d/yyyy format
        System.out.print(month + "/" + day + "/" + year);
```

```
    }

    // Print the date in long format
    public void printLongDate() {
        // Get the day, month and year
        int day = getDay();
        int month = getMonth();
        int year = getYear();
        // Show the date in long format
        System.out.print(day + " ");
        System.out.print(MONTH_NAMES[month - 1] + ", ");
        System.out.print(year + ".");
    }
}
```

DateHandler2.java

```
// Definition of class DateHandler2
public class DateHandler2 {
    // Attributes

    // An array for storing all month names
    private static final String[] MONTH_NAMES = {
        "January", "February", "March", "April",
        "May", "June", "July", "August",
        "September", "October", "November", "December"
    };

    // Methods

    // Show the details of a Date1 object
    public void saySomethingAboutDate(Date1 date) {
        // Check if the object is actually a BritishDate5 object
        if (date instanceof BritishDate5) {
            // It is verified that it is a BritishDate5 object
            System.out.print("The British date (");
            date.print();
            System.out.print(") means ");
            BritishDate5 bDate = (BritishDate5) date;
            bDate.printLongDate();
            System.out.println();
        }
        // Check if the object is actually an AmericanDate5 object

        else if (date instanceof AmericanDate5) {
            // It is verified that it is an AmericanDate5 object
            System.out.print("The American date (");
            date.print();
            System.out.print(") means ");
            AmericanDate5 aDate = (AmericanDate5) date;
            aDate.printLongDate();
            System.out.println();
        }
        // Check if the object is actually an ChineseDate5 object
        else if (date instanceof ChineseDate5) {
```

```
            // It is verified that it is an ChineseDate5 object
            System.out.print("The Chinese date (");
            date.print();
            System.out.print(") means ");
            ChineseDate4 cDate = (ChineseDate5) date;
            cDate.printLongDate();
            System.out.println();
        }
        // Otherwise
        else {
            System.out.print("The date (");
            date.print();
            System.out.print(") means ");
            System.out.print(MONTH_NAMES[date.getMonth() - 1]);
            System.out.print(" "  + date.getDay());
            switch (date.getDay()) {
            case 1: System.out.print("st"); break;
            case 2: System.out.print("nd"); break;
            case 3: System.out.print("rd"); break;
            default: System.out.print("th"); break;
            }
            System.out.println(", " + date.getYear() + ".");
        }
    }
}
```

TestDateHandler2.java

```
// Definition of class TestDateHandler2
public class TestDateHandler2 {

    // Main executive method
    public static void main(String args[]) {
      // Create a DateHandler2 object and is referred by variable
      // handler
      DateHandler2 handler = new DateHandler2();

      // Create a Date1 object
      Date1 date1 = new Date1();
      date1.setDay(1);
      date1.setMonth(2);
      date1.setYear(2001);
      // Call the DateHandler2 object with the Date1 object referred
      // by variable date1
      handler.saySomethingAboutDate(date1);

      // Create a BritishDate5 object
      Date1 date2 = new BritishDate5();
      date2.setDay(3);
      date2.setMonth(4);
      date2.setYear(2002);
      // Call the DateHandler2 object with the BritishDate5 object
      // referred by variable date2
        handler.saySomethingAboutDate(date2);

      // Create an AmericanDate5 object
```

```
      Date1 date3 = new AmericanDate5();
      date3.setDay(5);
      date3.setMonth(6);
      date3.setYear(2003);
      // Call the DateHandler2 object with the AmericanDate5 object
      // referred by variable date3
       handler.saySomethingAboutDate(date3);

      // Create an ChineseDate5 object
      Date1 date4 = new ChineseDate5();
      date4.setDay(7);
      date4.setMonth(8);
      date4.setYear(2004);
      // Call the DateHandler2 object with the ChineseDate5 object
      // referred by variable date4
       handler.saySomethingAboutDate(date4);
   }
}
```

Self-test 7.15

1 **GeneralStaff.java**

```
// Definition of class GeneralStaff
public class GeneralStaff {
    // Attributes
    private int category;
    private String name;
    private double basicSalary;
    private double salesVolume;
    private double commissionRate;
    private double allowance;
    private double hourlyWage;
    private double workingHours;

    // Constants for different staff categories
    public static final int CLERK = 1;
    public static final int SALESPERSON = 2;
    public static final int MANAGER = 3;
    public static final int TEMP_STAFF = 4;

    // Constructor
    public GeneralStaff(int category, String name,
            double basicSalary, double salesVolume,
            double commissionRate, double allowance,
            double hourlyWage, double workingHours) {
        this.category = category;
        this.name = name;
        this.basicSalary = basicSalary;
        this.salesVolume = salesVolume;
        this.commissionRate = commissionRate;
        this.allowance = allowance;
        this.hourlyWage = hourlyWage;
        this.workingHours = workingHours;
    }
    // Get the staff name
```

```
    public String getName() {
        return name;
    }

    // Find the salary
    public double findSalary() {
        switch (category) {
        case CLERK:
            return basicSalary;
        case SALESPERSON:
            return basicSalary + salesVolume * commissionRate;
        case MANAGER:
            return basicSalary + allowance;
        case TEMP_STAFF:
            return hourlyWage * workingHours;
        default:
            return 0.0;
        }
    }

    // Find the net salary after MPF contribution
    public double findNetSalary() {
        double salary = findSalary();
        if (salary < 5000.0) {
            return salary;
        }
        else if (salary >= 20000.0) {
            return salary - 1000.0;
        }
        else {
            return salary * 0.95;
        }
    }

    // Find the MPF contribution by the company
    public double findMPFByCompany() {
        return (findSalary() > 20000.0) ? 1000.0 : findSalary() * 0.05;
    }

    // Find the MPF contribution by the staff
    public double findMPFByStaff() {
        double salary = findSalary();
        if (salary < 5000.0) {
            return 0.0;
        }
        else if (salary >= 20000.0) {
            return 1000.0;
        }
        else {
            return salary * 0.05;
        }
    }
}
```

PayrollCalculator2.java

```
// Definition of class TestPayrollCalculator2 {
public class TestPayrollCalculator2 {

    // Main exeuctive method
    public static void main(String args[]) {
        // Create an array of staff in the company
        GeneralStaff[] staff = {
            new GeneralStaff(GeneralStaff.CLERK,
                "Mary", 4000.0, 0.0, 0.0, 0.0, 0.0, 0.0),
            new GeneralStaff(GeneralStaff.CLERK,
                "Peter", 6000.0, 0.0, 0.0, 0.0, 0.0, 0.0),
            new GeneralStaff(GeneralStaff.SALESPERSON,
                "Joe", 4000.0, 100000.0, 0.05, 0.0, 0.0, 0.0),
            new GeneralStaff(GeneralStaff.SALESPERSON,
                "Amy", 5000.0, 200000.0, 0.08, 0.0, 0.0, 0.0),
            new GeneralStaff(GeneralStaff.MANAGER,
                "John", 20000.0, 0.0, 0.0, 10000.0, 0.0, 0.0),
            new GeneralStaff(GeneralStaff.TEMP_STAFF,
                "Benny", 0.0, 0.0, 0.0, 0.0, 50.0, 160.0)
        };

        // Create a PayrollCalculator2 object by supplying an array
        // of Staff objects
        PayrollCalculator2 calculator = new PayrollCalculator2(staff);

        // Show the payroll report
        calculator.showReport();
    }
}
```

2 The required modifications to the source code listings in Self-test 7.10 and this self-test are highlighted. You can see that the changes required for Self-test 7.10 are to create a new subclass `TempStaff` and add a statement in the `main()` method of the `TestPayrollCalculator1` class. However, the changes required for this self-test are to add new attributes and new constants. The constructor parameter needs modification, and the `switch/case` statement has to handle a new category.

The necessary modification for Self-test 7.10 is localized and just to create a new subclass, whereas changes are needed in various parts of the `GeneralStaff` class in this self-test. Any missing or inconsistent change to the `GeneralStaff` class can give wrong results, which is why such implementation is said to be more error-prone. Therefore, the implementation used in Self-test 7.10 is much better with respect to software extensibility and maintenance.

The two implementations illustrate the usage of an abstract superclass with concrete subclasses and the beauty of polymorphism.