MT201

Unit 3

Classes and objects

Course team

Developer:	Herbert Shiu, Consultant
Designer:	Dr Rex G Sharman, OUHK
Coordinator:	Kelvin Lee, OUHK
Member:	Dr Reggie Kwan, 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

In *Unit 2*, you learned how to analyse a problem and to derive the objects involved. From such an analysis, you can find the attributes and behaviours of each set of objects in an appropriate way. If you can model those objects in programs and manipulate the objects' states, you can solve the problem. Please review *Unit 2* if you are not clear on how this is done.

As long as the objects can operate in the way you predefined, you will be able to solve a problem. It's just like being a manager in an office. You instruct your subordinates to complete the tasks you assign to them. They might work together or alone to complete the tasks. The most important point here is that you don't have to dig into the details of their work and tell them every step they need to take. At a managerial level, you just need to have an overview of what your subordinates are doing. Of course, you might have to teach them how to perform some tasks before they start the work.

In solving a problem with your programs, you are the manager of the objects. Your ultimate goal is to manipulate the objects and solve the problem in the way you desire.

There may be many objects involved in your problem, and it is almost impossible to define each individual object. Fortunately, by investigation, you can classify the objects into a few groups so that each group of objects has the same set of attributes and behaviours. In software, you can define a class to describe a group of objects.

This unit teaches you how to define classes, their attributes and behaviours in Java. It is the first step in the design phase. Once you have the objects ready, you can further elaborate the objects' behaviours.

Get ready and here we go!

Objectives

At the end of *Unit 3*, you should be able to:

1. *Define* Java classes, attributes and methods.
2. *Use* Java identifiers.
3. *Use* objects in Java programs.
4. *Use* dot notation to access the object members.
5. *Name* and *classify* the eight Java primitive types.
6. *Differentiate among* the primitive types and classes.
7. *Select* the suitable primitive type for different data.
8. *Describe* the conversions between primitive types.
9. *Perform* numeric calculations.
10. *Describe* the uses and importance of comments.
11. *Use* class packages.
12. *Differentiate among* instance members and class members.

Defining classes in Java programs

Unit 2 discusses how to solve a real-world problem by manipulating the relationships and operations of objects. Alternatively, you can achieve the same result if you can model such relationships and operations by using software. By writing Java programs, you model those objects, their attributes and behaviours, and the relationships among them.

In a Java program, a class declaration describes a particular type of object that has the same set of attributes and behaviours. An analogy to this is constructing a building. The architect designs a blueprint, and a building company constructs it. The blueprint is like the class declaration, and the building constructed is like the object created while you run your program. The building company can use the same blueprint to construct many buildings with the same structure. Similarly, you can use a single class declaration to create many objects of the same type.

The way to declare a class in a Java program is:

```
class class-name {
   variable-declarations
   method-declarations
}
```

The declaration is presently incomplete, and you will revisit this declaration later in the unit.

The above declaration provides three types of information:

a *class-name*

b *variable-declarations*

c *method-declarations*.

The *class-name* is the name you call this group of objects in your problem. For example, if your problem involves bank accounts, you would probably name the class of this type of objects `Account`. That is:

```
class Account {
.. .. ..
}
```

Variable-declarations and *method-declarations* correspond to the attributes and behaviours of the objects respectively. We discuss them in detail soon. The order of variable declarations and method declarations is arbitrary, and you can even mix them. Some programmers prefer to place the variable declarations before the method declarations to highlight classes that are data centric. For classes that are behaviour centric, in which the core considerations of the classes are their behaviours, some programmers prefer to place the method declarations before the variable declarations. You can develop your own conventions when you gain more programming experience with Java.

The use and importance of comments

When you write your Java program with a text editor such as Notepad, you are quite clear about what you are writing. You edit the program, compile it and execute it repeatedly until you are satisfied with the output of the program. Then, you leave the program file in the hard drive of your computer, as if you will never use it again.

Some time later, you find that you need to run the program again. You start your computer and run the same program again. It works perfectly and you are pleased with the program.

However, the world is changing rapidly, and you find that your previously written program no longer fulfils your needs. You find that your program fulfils nine-tenths of the situations and it is necessary to modify your program to cater for the remainder. You start an editor and read the program from the hard drive, and you will probably find that you no longer understand what you previously wrote. You have two choices. One is to study the program thoroughly to determine what it really does, and the second one is to write another one to replace it.

Another similar situation is working with your classmates to develop a software application in a project. Every team member is responsible for developing part of it, and the parts are integrated later to produce the final software application. You are no longer alone and you need to understand the programs written by your team members. Then, you probably find that you have to study what they are writing. As everyone usually uses a different style in writing programs, the result is that you find understanding their work is even more difficult than studying your own programs.

Therefore, it is always good practice to add some remarks or comments in the program to make it self-explanatory. The Java compiler will simply ignore the comments — you shouldn't.

The way to add comments in your Java program is:

```
/*
   This is the class definition of a bank account
 */
class Account {
......
}
```

The block enclosed by the '`/*`' and '`*/`' is ignored by the compiler, and you can therefore write down the comments to make your program self-explanatory.

The following is a list of recommendations for using comments:

1 You should add comments to class definitions explaining their uses, such as the real-world objects they are modelling.

2 Comments for variables can tell the readers what values they will store.

3 Adding comments to methods can describe their functionalities.

4 Lengthy methods can be difficult to understand. If you write long methods, you should place comments before blocks of statements that perform unique operations.

Using the '`/*`' and '`*/`' pair, you can write down single-line and multiple-line comments. If you want to add a comment for each statement, for example,

```
statement1;     /* This is the comment for statement1 */
statement2;     /* This is the comment for statement2 */
statement3;     /* This is the comment for statement3 */
```

Java provides another convenient way to add single-line comments in your program:

```
// This is a single-line comment starting in the first column
statement1;     // This is the comment for statement1
statement2;     // This is the comment for statement2
statement3;     // This is the comment for statement3
```

The double forward slashes (`//`) in the above program segment represent the start of a comment. Everything after the slashes is considered to be comments, and the compiler will just overlook them.

Naming classes and class members

To ease yourself into relating the class declarations to real-world objects, you should name the classes and their members, including variables and methods, carefully. In the example in the previous section, the class for all bank accounts is named `Account` as a natural and logical consequence. The following subsection gives you further details.

Identifiers

The names of classes, variables and methods are known as *identifiers*. Simply, a word like `Account` can be an identifier. However, there are some restrictions or rules when designing identifiers. The following reading helps you understand the restrictions and provides recommendations for you.

Reading

King, Section 2.6, pp. 44–45

The reading gives you a set of general guidelines for choosing identifiers. To summarize the rules for you, *identifiers* in Java may consist of any letters, digits and the underscore character. They must start with a letter or underscore character, and there is no limit on the length.

Furthermore, the reading provides you with two naming conventions for variables and classes respectively. I would like to provide further information about these conventions.

The identifiers of method names are usually verbs, because methods correspond to the object behaviours (what they do). For example:

```
deposit()
setMinLimit()
```

Identifiers for variables are usually nouns. For example:

```
currentBalance
minLimit
```

As you can see, we usually do not use the underscore character (_) for identifiers. If more than one word is in the identifier, the subsequent words should start with an uppercase letter.

Class identifiers should start with uppercase letters, such as `Account` and `SavingsAccount`. All variables should start with lowercase letters, such as `interest` and `accountBalance`.

Furthermore, Java keywords cannot be used as identifiers. You can find all of the Java keywords in the following section.

Keywords in Java

Keywords are the building blocks of the Java language, and each has its own meaning. Therefore, you cannot use these keywords as identifiers.

You can find out all of the keywords and three reserved words in Java from the following reading.

Reading

King, ‘Keywords’, pp. 45–46. You might also attempt the review questions on p. 46.

In Table 2.1 on page 45 of the textbook are 47 keywords. For completeness, two keywords, `strictfp` and `assert`, were added to Java language since versions 1.2 and 1.4 respectively.

Furthermore, here are a few observations from the keyword list:

1. All keywords and reserved words are in lowercase. As identifiers in Java are case-sensitive, any keyword with any character capitalized is no longer a keyword but a valid identifier.
2. `const` and `goto` are keywords in Java but they are no longer used. Such an arrangement restricts programmers from using these two words as identifiers, because most programmers have preconceived ideas about what these two words mean.
3. `true`, `false` and `null` are not keywords but are pre-defined literals (i.e. pre-defined constant values), which means that they represent special values in Java.

Self-test 3.1

Please determine whether the following entities are valid identifiers:

mt201	java_is_easy	balance4Account	12aDozen
interface	AccountBook	Class	

Defining attributes

Each object has its own attributes and states, and you can use variable declarations to define those attributes. A possible way to declare a single attribute in a class definition is:

type identifier;

For example, you might use

```
double balance;
```

to define an attribute `balance` with type `double`. (We'll discuss types very soon.)

If you have a list of attributes of the same type, you can use a single statement to declare that list of attribute identifiers by delimiting them with commas. For example:

```
double balance, overdrawnLimit;
```

In a computer program, a variable represents a memory location in the computer that stores a value. We usually call the variables, such as `balance` and `overdrawnLimit` in the example above, the *instance variables* of the class, because when you run the program and create an object based on the class definition, the computer allocates memory locations for these attributes for each object to store the attribute values (or states).

We discussed identifiers in the previous section in this unit, but how about the types?

Basic types — `int` and `double`

Let's consider a vehicle object. Each vehicle has a maximum passenger capacity limit and a fuel consumption rate as its attributes. You can declare this `Vehicle` class and define these two attributes. In real life, these two attributes would be numeric, and we could handle them easily in any calculations. However, this is not so for computers.

Use the following reading to understand the way computers store data.

Reading

King, Section 1.4, pp. 13–14

You now know that computers handle integral values (whole numbers or integers) and floating-point values (that is, values with fractions) differently because they are stored in different formats in computers. This issue is discussed in greater detail later.

For the time being, we discuss the two most commonly used numeric data types in Java — `int` and `double`.

You can use `int` as the variable type so that the variable can store any integral value. If you want to store a floating-point value, you should use `double`.

Maximum capacity limit (the number of passengers) must be integral (an integer), and fuel consumption rate probably is not a whole number. By combining the type and the identifier, we can define the maximum capacity and the fuel consumption rate attributes as:

```
int maxCapacity;
double fuelConsumptionRate;
```

By placing these two attributes in the `Vehicle` class definition, we get:

```
// The class definition for a vehicle
class Vehicle {
  int maxCapacity;            // Maximum passenger capacity
  double fuelConsumptionRate; // Average fuel consumption rate
}
```

Although we have not provided any method declaration, the above class definition is valid. Use the following self-test to make sure that you really understand the way to define classes with attributes only.

Self-test 3.2

1 Define a class for dates with day, month and year attributes.

2 Define a class for student examination results including the subjects of Chinese, English and Mathematics.

3 Define a class for computers with CPU speed, hard-disk size, memory size, CD-ROM speed and price as attributes.

Defining methods

Every object has its own attributes and behaviours. We have discussed how to define instance variables as object attributes. To model the behaviours, you can define a method to represent each particular operation of the object.

Instance methods as object behaviours

For example, you need to model a ticket counter object in your problem, and you find that a ticket counter has an attribute `reading` and a behaviour `increase`. Without the behaviour, the ticket counter can be declared in your program as:

```
// The class definition for a ticket counter
class TicketCounter {
  int reading;        // The ticket counter reading
}
```

A way to declare a behaviour in your program is:

```
return-type method-name() {
  statements
}
```

You provide a *return-type* if you need an instant result after performing the behaviour. We provide an example to demonstrate a behaviour that has an instant result, later in this unit. For the mentioned behaviour, `increase`, you do not need to get any result from it. Therefore, you use a keyword `void` to signify that the method returns no result.

You can define the `increase` behaviour as:

```
void increase() {
  reading = reading + 1;
}
```

In the above method declaration, there is only one statement:

```
reading = reading + 1;
```

In Java, any statement can span more than one line but must be terminated by a semi-colon (`;`).

When you run your program and there is a ticket-counter object, the ticket counter has the instance variable `reading` with a particular value. For example, its value is zero by default at the beginning.

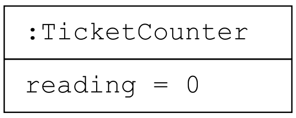


Figure 3.1 A ticket counter with a zero `reading` value

The above diagram (Figure 3.1) helps you visualize an object. There are two regions in the above diagram. The upper one contains the class name, and the prefix colon (:) means it is an object. In this situation, it is an object of the class `TicketCounter`. The lower region contains the attributes and their current values. In the above diagram, there is only one instance variable, `reading`, and it is storing a value of 0. It is a simplified version. You'll see a more complete diagram later in the unit.

You can use the expression

```
reading + 1
```

to get the result of the instance variable `reading` plus one. In the statement

```
reading = reading + 1;
```

the left-hand side of the equals sign must be a variable that can store the result of the expression on the right-hand side (see Figure 3.2). The function of the equals sign is to *assign* the value of the expression on the right-hand side to the variable on the left-hand side. For the above statement, the variable that stores the expression result is the instance variable `reading`.

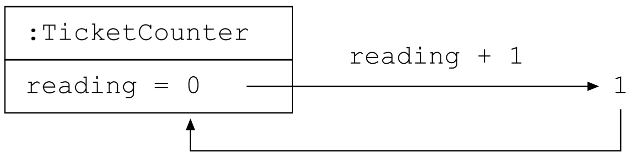


Figure 3.2 One is being added to the ticket counter's original reading value 0

That is, the original instance variable `reading` is used to get the expression result, and the expression result will replace the value in the attribute `reading`. You can see that after performing the above behaviour, the value of the instance variable `reading` is increased by one (Figure 3.3).

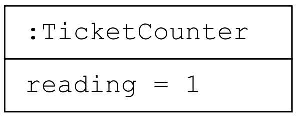


Figure 3.3 The result after the addition

By inserting the method declaration into the aforementioned class, you get:

```
// The class definition for a ticket counter
class TicketCounter {
  int reading;        // The ticket counter reading

  // To increase the reading by one
  void increase() {
    reading = reading + 1;
  }
}
```

You can now observe from the class definition that the `TicketCounter` class has one attribute `reading` and one behaviour `increase`. During run-time, you can use the above class definition to create an instance of the class `TicketCounter`. (The way to create an object at run-time is discussed later in this unit.)

In the `increase` method, the instance variable `reading` corresponds to the value of the ticket counter object. Then, every time the object performs the `increase` behaviour, the object's attribute, `reading`, is increased by one. It is a significant observation that every time an object performs an operation or behaviour, the operations are applied on its own attributes.

Return values

Sometimes, you might need to know the result or the value of an attribute after performing some behaviours. For example, if you were a manager in an office and you assigned a task to your subordinates, say, getting the faxes from the fax machine, you might want to know how many new faxes your subordinates got.

For such scenarios, you can define a method with a return type, such as the types `int` and `double`, discussed earlier, so that a value will be returned.

For example, if you want the `TicketCounter` object to tell you the current value of `reading`, you can define a behaviour `getReading` for the class, like:

```
int getReading() {
  return reading;
}
```

Compared with the previous method, the differences are only that there is a return type (not `void`) and a `return` statement. You can still have other statements preceding the `return` statement, but there is usually no statement after it. You use the `return` statement to return a value after performing a behaviour.

By adding the `getReading()` method, the class for ticket counter becomes:

```
// The class definition of a ticket counter
class TicketCounter {
  int reading;        // The ticket counter reading

  // To increase the reading by one
  void increase() {
    reading = reading + 1;
  }
  // To get the ticket counter reading
  int getReading() {
    return reading;
  }
}
```

In the above method, `getReading()`, the value to be returned is the content of the instance variable `reading`. This means that you can get the attribute value (or state), `reading`, of a ticket-counter object. Please bear in mind that the value to be returned must match the return type preceding the method name in the method declaration. With respect to the above `getReading()` method, the return type is the `int` type because the value to be returned is the content of instance variable `reading`, which is also an `int` type.

Local variables

In the previous section, you saw that instance variables are the attributes of an object. Practically, they represent a storage location in the computer's memory for storing values. If you want to store some values while you run your program, you need variables.

A situation in which you need variables is the time an object is performing its behaviours, and it needs some temporary locations for storing intermediate values. For example:

```
// The class definition of a circle
class Circle {
  double diameter;     // The circle’s diameter

  ……
}
```

Based on the above class definition, `Circle`, the objects created based on it only have one attribute, `diameter`. To calculate the area of the circle, it is preferable to determine its radius first. To write the method, `calculateCircleArea()`, it is therefore necessary to have a temporary storage to store the radius value. For example:

```
// The class definition of a circle
class Circle {
  double diameter;     // The circle diameter

  ……

  // To calculate the circle area
  double calculateCircleArea() {
    double radius = diameter / 2.0;
    return 3.1415 * radius * radius;
  }
}
```

In a method, you can declare some temporary variables, known as local variables, by using the following syntax:

type identifier;

The syntax is the same as declaring instance variables. The main difference is that the local variable declaration is placed within a method definition, whereas an instance variable definition is placed within a class definition outside of all method definitions.

Another practical difference between instance variables and local variables is that local variables are created when the method is executed. When the computer completes the method execution, it automatically removes these variables. By contrast, instance variables last as long as the object that contains them.

Method parameters

An object may perform a behaviour under different situations. For example, if you were a manager in an office, you might give piles of documents to your subordinates for making photocopies. To provide different situations or data to the object so that it can perform accordingly, you can provide a parameter list in the method declaration. For example, you can add a new operation to the ticket counter that can increase the counter by a value other than one in the following way:

```
void increaseByAmount(int amount) {
  reading = reading + amount;
}
```

Compared with the previous `increase()` method, there are two main differences:

1 There is a parameter list (here, a list of only one parameter), `int amount`, in the method declaration. If you want the object to perform such a behaviour, you have to provide values to fulfill the parameter list. For example:

```
increaseByAmount(2)
```

Then, the object gets a value of two from the parameter, `amount`, and performs the behaviour. Therefore, you can use this way to instruct the object to perform its behaviour under different situations.

The expression is changed to

```
reading = reading + amount;
```

which means the sum of the values of `reading` and `amount` is used to update the instance variable `reading`.

Now, the class declaration for ticket counter becomes:

```
// The class definition of a ticket counter
class TicketCounter {
  int reading;        // The ticket counter reading

  // To increase the reading by one
  void increase() {
    reading = reading + 1;
  }

  // To increase the reading specified by the parameter, amount
  void increaseByAmount(int amount) {
    reading = reading + amount;
  }

  // To get the ticket counter reading
  int getReading() {
    return reading;
  }
}
```

The order of the method declaration is arbitrary. It’s up to you to use your own conventions.

2 If you want to specify more than one aspect of the situation in which the object performs its behaviour, you can specify a parameter list with more than one parameter. For example, a method to deposit money to an account in Hong Kong dollars using different currencies could be:

```
void deposit(double amount, double exchangeRate) {
  balance = balance + amount * exchangeRate;
}
```

By combining all the examples we discussed above, we can get a general format to declare a method:

return-type method-name(*parameter-list*) {
  variable-declarations;
  statements
}

You should use the following self-test to ensure that you have a thorough understanding of the way to define a class.

Self-test 3.3

Define a class `Account` with an attribute `balance` and two behaviours, `deposit` and `withdraw`.

Manipulating objects

In the previous section, we mentioned that you can use a class definition to model a class of similar objects in the real world, and we discussed the way to define a class in Java programs. Now everything is ready and we can talk about the way to model a real-world situation.

In the real world, if you want to have a ticket counter to count tickets or anything else, you can buy one from a department store and keep it in your home. Whenever you want to use it, you have to find it and hold it in your hand and press the trigger to increase the count.

Similarly, if you want to have an imaginary ticket counter while you run your program, you first have to get one — or, more exactly, to create one. Then, you have to find a way to locate it and instruct it to do something for you.

From the above paragraph, you can see that the steps to manipulate an object are:

1 Create an object.

2 Have a way to find it.

3 Give instructions to it.

The following subsection discusses these three aspects in detail.

Creating objects

Let's consider a real-life scenario.

You buy a ticket counter from a department store, bring it home and keep it somewhere in your home. You suspect that some time later you might forget where you stored it, so you wisely write a note on a piece of paper to help you locate the newly bought ticket counter. For example, the note might read:

> The ticket counter is stored in the first drawer of the cabinet in the left-hand corner in the storeroom.

Then, you put this piece of paper in a little box and stick a label 'Ticket Counter' on the cover. As you have many such little boxes for reminding yourself where you put your accessories, you place all these boxes together so that you can find them easily. Whenever you want to use the counter to count something, you use your little box to remind yourself where the ticket counter is, and then you get it and use it.

The above is what you might do in real life. Now, let's see if you can do the same thing in your program.

First of all, you need a way to find your ticket counter object. To prepare a ‘little box’ that stores the description to find your ticket counter object in your program, you use the following statement:

```
TicketCounter counter;
```

The `counter` in the above statement is a variable in your program. It is usually known as a *reference variable* because you use it to *refer* to an object when you run your program. You can treat it as a little box with a label (`counter`) on its cover and it can tell you how to find your ticket counter. In Java programming, the way to find or get an object is known as a *reference* or *handle,* which is the reason why the variable is called a reference variable.

You can visualize the above situation by using the following diagram (Figure 3.4):

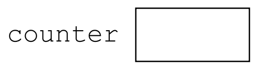


Figure 3.4 A reference to a ticket counter

You should notice that preparing a box does not mean that you already have a ticket counter. You still need to find some ways to have one. It is not necessary for you to worry about how these boxes (variables in your program) are stored. That’s the computer’s responsibility.

In software, you cannot buy a ticket counter — you must create one. As we mentioned that a class definition is a blueprint of a group of objects, you can use the following syntax to create an object based on a class definition:

```
new TicketCounter()
```

`new` is a Java keyword that you can use to create an object based on the class that follows it. The parentheses that follow the class name, `TicketCounter`, are mandatory (they must be there). The significance of these parentheses is discussed in *Unit 7*.

The above statement creates a new ticket-counter object based on the class `TicketCounter`. This object is imaginary but has all the attributes and behaviours that you defined in the class definition. For example, based on the `TicketCounter` class definition on page 14, the ticket counter object has an attribute, `reading`, and three behaviours, `increase`, `increaseByAmount` and `getReading`.

You can now imagine that the following object exists in your computer (see Figure 3.5):

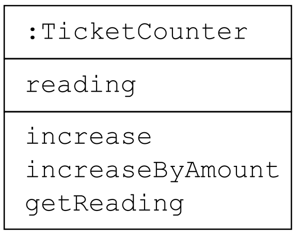


Figure 3.5 A ticket counter object

The above is the complete diagram of the one we discussed on page 10. We use the above notation in the course to represent an object while you run your program. The topmost row stores the class name of the object. The second and third rows are the object's attribute and behaviours respectively.

Besides creating an object, the keyword `new` tells you where you can find the object. That is the reference to the newly created object. In your program, you can use the following statement to place the description that can find the object in the box labeled 'counter' immediately after the object is created:

```
counter = new TicketCounter();
```

By combining the statements, you have the following program segment:

```
TicketCounter counter;
counter = new TicketCounter();
```

The following diagram (Figure 3.6) should help you visualize the whole picture:

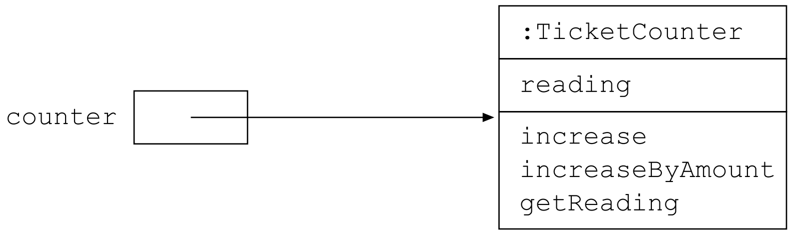


Figure 3.6 A variable `counter` references a ticket counter object

The arrow in the above diagram means there is a way (or reference) in the variable `counter` that can refer to an actual object of class `TicketCounter`.

As a summary, a generic way to create an object is:

Class-name variable-name;
variable-name = `new` *Class-name();*

For convenience, you can combine the above two statements into one:

Class-name variable-name = `new` *Class-name();*

The above single statement performs the following actions:

1 prepares a box (declaring a variable)

2 buys an object (creating an object)

3 stores the description that describes the location of the object into the box (assigning the way to find the object into the variable).

Therefore, the example of the ticket counter can be written in a statement like:

```
TicketCounter counter = new TicketCounter();
```

You should bear in mind a few differences in the situations in real life and in writing your programs.

1 You can place two pieces of paper that tell you the ways to find two individual accessories in the same little box. However, a variable in your program can only store a single entity, which means that if you have two ticket counters in your program, you need two variables for referring to them. Otherwise, when you place the second reference for the second ticket counter in a variable that already has a handle, you lose the first one.

2 You can stick two identical labels on two different boxes, but the contents are actually referring to two different accessories. However, you cannot use two variables with the same name. Otherwise, the computer does not know which one you are actually using.

3 If later on you don't need to find the ticket counter any more, you can put another piece of paper in the box that describes the way to find another accessory, say a hair dryer, and stick another label, 'Hair Dryer', on the cover. In Java programming, you cannot rename a variable, but you can assume that you have infinite number of such variables.

4 If you lose the description in the box, you can still search through your home and you'll certainly find the ticket counter — eventually (provided that you don't have a pet that opens drawers and throws things out of the window). But, if you lose the reference to an object in your program, you can no longer find it.

5 To help yourself to find different types of accessory, you might use colours to categorize them. For example, you might use red boxes to store the descriptions for stationery and blue ones to store those for computer accessories. Of course, if you run out of red boxes, you can use a blue box to store the description to find a ruler. In programming, it is somewhat similar and somewhat different. Each variable similarly has a name (the box label) and a type (the box colour). However, you cannot place a reference for a `BankAccount` object into a variable of `TicketCounter` type.

There are several variations that you may find useful.

If you need to declare a few variables at the same time, you can use the following syntax for convenience:

Class-name variable-name1, variable-name2, …;

For example,

```
TicketCounter counter1, counter2;
```

The `null` literal

If you want to prepare a little box for storing the map that will lead you to get a floppy diskette, you might stick a label, 'Floppy Diskette', on the box. If you don't have the floppy diskettes yet, you place a message, 'To be purchased', in the box.

Similarly, if you want to prepare a variable that is going to store a reference to an object, you can also assign a special content that is more or less similar to the 'To be purchased' message in the scenario above. This special content is known as `null` in the Java programming language. Therefore, if you want to indicate a variable that refers to nothing, you can use the following statement:

```
counter = null;
```

If the `counter` variable was storing a reference to an object, it loses that handle now.

Implicit and explicit initialization

If you are developing a program to model the operations in a bank, you definitely need to model bank accounts. Every bank account has at least an attribute `balance` and you therefore write the following Java program:

```
// The class definition for a bank account
class Account {
  double balance;       // The account balance
}
```

If you open a bank account in a bank, the account balance is initially zero. Similarly, if you create an object based on the above class definition `Account`, the object attribute `balance` is initially zero. This is known as *implicit initialization.*

If the bank provides a pre-defined overdrawn allowance to all customers, a customer can overdraw money from his or her account, and the overdrawn allowance is reduced accordingly. To model such a situation, each `Account` object must have its own overdrawn allowance and a behaviour `overdraw`. The definition of the class `Account` becomes:

```
// The class definition for a bank account
class Account {
  double balance;              // The account balance
  double allowance = 10000.0;  // Pre-defined overdrawn allowance
  // To overdraw an account
  public void overdraw(double amount) {
    allowance = allowance – amount;
  }
}
```

The above class definition is incomplete because it is necessary to verify the overdrawn allowance before overdrawing a bank account. *Unit 4* discusses how to implement verification. The above program is just for illustration.

In the class definition above, you can see that the definition of the instance variable `allowance`,

```
  double allowance = 10000.0;
```

is followed by an equals sign and a value. This statement declares a variable `allowance` of `double` type and assigns a value of 10000.0 to it when the `Account` object is created. Initializing a variable in this way is known as *explicit initialization*.

According to the class definition above, when you create an `Account` object in your program, the object has two attributes, `balance` and `allowance`. The value initially stored in the attributes `balance` and `allowance` is 0.0 and 10000.0 based on implicit and explicit initialization respectively.

Creating more objects of the same class

We have discussed using a class definition to model a group of similar objects that have the same sets of attributes and behaviours. You can create objects based on the same class definition — as many as necessary. Every time you use the keyword `new` to create an object, it is a new object that has its own sets of attributes and behaviours.

For example, if you need two ticket-counter objects in your program to count objects from two categories, you can create two ticket counters:

```
new TicketCounter();  // Create the first counter
new TicketCounter();  // Create the second counter
```

Usually, you need ways to find these two ticket-counter objects, and you therefore declare two variables to store the ways to get them:

```
// Create the two counters and stored them in variables
// counter1 and counter2 respectively
TicketCounter counter1 = new TicketCounter();
TicketCounter counter2 = new TicketCounter();
```

An alternative way to declare two variables and create two objects is:

```
TicketCounter counter1 = new TicketCounter(),
              counter2 = new TicketCounter();
```

Practically, there are two `TicketCounter` objects in the computer's memory, which can be located by the two variables (see Figure 3.7).

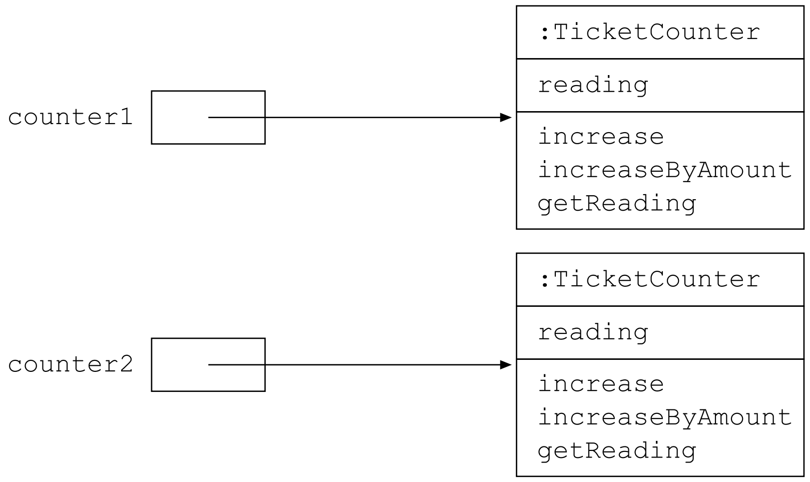


Figure 3.7 Two variables referring to two different ticket counter objects

There are two different ticket-counter objects. Each has its own variable `reading,` which should contain different values, and three methods, `increase`, `increaseByAmount` and `getReading`. It is as if there are two ticket counters in the real world, each of which contains and looks after its own reading.

Accessing object members

You can now create objects and find them by using variables whenever necessary. Let's go on to discuss the next step in manipulating objects — accessing object members — which are variables and methods of an object. You access the object members to update the object states to reflect the changes in the real world.

Object members include the variables (attributes) and methods (behaviours) that an object owns — for example, `reading`, `increase`, `increaseByAmount` and `getReading` — all of which are members of an object of the class `TicketCounter`. The former is a variable and the latter three are methods.

To access a member of an object, you may use dot-notation:

reference.member

You can use a reference variable to provide the *reference* and the member can be any variable or method defined in a class.

For example, if you have the following statement in which the variable `counter` stores the reference to the newly created object,

```
TicketCounter counter = new TicketCounter();
```

you can use the notation below to refer to the variable `reading` of the object:

```
counter.reading
```

An analogy is that you use a little box labeled '`counter`' to find a ticket- counter object. Then, you are going to manipulate an attribute `reading` in it.

For example, if you want to assign a value to `reading`, you can do this as:

```
counter.reading = 10;
```

The above statement stores a value of 10 in the instance variable (the attribute) `reading` of an object that is referred by the variable `counter`. After the above statement is executed, the content of the `reading` variable of the object that `counter` is referring to is 10.

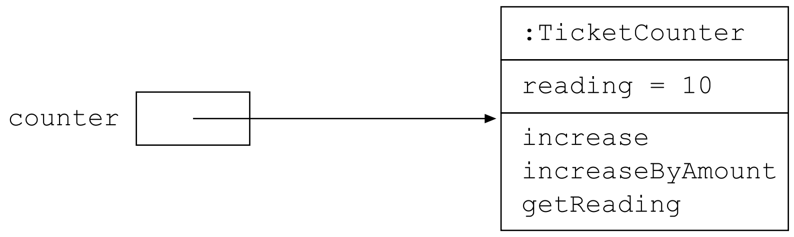


Figure 3.8 The state of a ticket counter after 10 is assigned to `reading`

It is always preferable not to assign a value directly to an object attribute but to send a message to the object so that it can maintain its own attribute values. For example, if you have the following statement in your program,

```
counter.reading = -1;
```

the compiler will always accept it and a value of –1 would be stored in the `reading` variable of a ticket-counter object, which implies that there is a ticket counter with a reading of –1 in the real world. That is unreasonable. The occurrence of such a situation in your program means that there must be some problems. The main cause of such a situation is that object attribute values were updated directly by some other objects. This is highly undesirable and you should avoid writing your program in this style.

By contrast, sending a message to an object so that it can update its own attribute is always preferable because the object knows exactly how to manipulate its own attributes. For example, if the ticket counter has an attribute `reading`, there are usually two corresponding methods — one for updating a value to the attribute and another one for retrieving the attribute value. We can further improve the definition of the `TicketCounter` class as:

```
// The class definition of a ticket counter
class TicketCounter {
  int reading;      // The ticket counter reading

  // To increase the reading by one
  void increase() {
    reading = reading + 1;
  }

  // To increase the reading specified by the parameter, amount
  void increaseByAmount(int amount) {
    reading = reading + amount;
  }

  // To get the ticket counter reading
  int getReading() {
    return reading;
  }

  // To set a reading to the ticket counter reading
  void setReading(int newReading) {
    reading = newReading;
  }
}
```

Then, if you want to change the attribute `reading`, you should use the following statement:

```
counter.setReading(10);
```

Unit 4 discusses how to implement verifications in methods and how a value is verified before it is assigned to an object attribute.

Let’s reconsider the scenario of two ticket counters created by the following statement:

```
TicketCounter counter1 = new TicketCounter(),
              counter2 = new TicketCounter();
```

If the statements below follow the above statement,

```
counter1.setReading(10);
counter2.setReading(20);
```

it means that the values of 10 and 20 are assigned to the variable `reading` of two ticket-counter objects that are referred to by variables `counter1` and `counter2` respectively. That is:

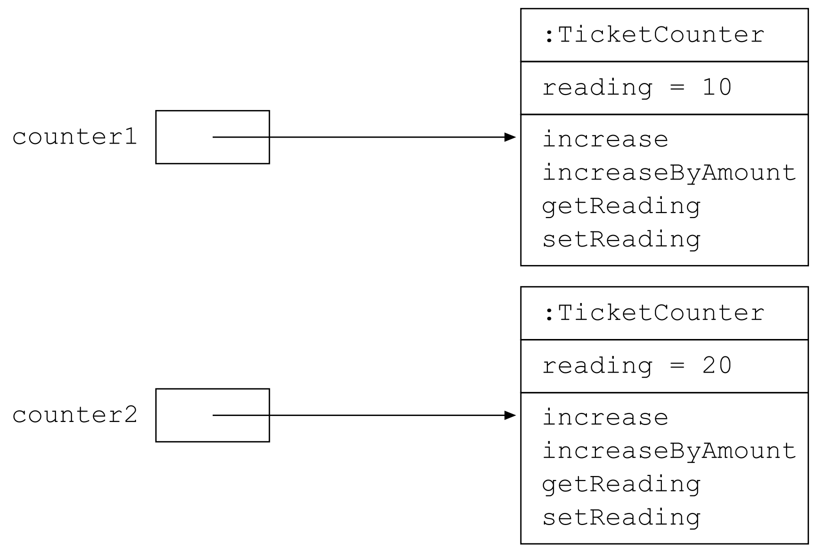


Figure 3.9 After assigning 10 and 20 to two ticket counters through the method setReading()

An analogy is — identical twins are born identical. They are exactly the same when they were born but then different living environments shape them with different characters. That is, the twins are individual entities but they can have different characteristics and attributes. If one of them changes his or her attribute, the other one will not necessarily change.

Now, if you received a gift that was very important, you might write down in two places the location where you stored it— in your Personal Digital Assistant (PDA) and in your diary. By doing so, you can find it by reading your PDA or diary.

Using the little boxes to illustrate this idea, you could have two boxes with labels 'Gift 1' and 'Gift 2' but they contain the same location information. You will find the gift by reading either one of them. If you want to do this, you first prepare two little boxes with different labels. Then, you write down the way to find your gift and store it in the first one. Then, you copy the same location information to another piece of paper and store it in the second box.

Similarly, if you have two variables and one ticket counter, you can use the following program section to do so:

```
TicketCounter counter1, counter2;
counter1 = new TicketCounter();
counter2 = counter1;
```

The meanings of the above statements are:

1 The first statement prepares two variables, `counter1` and `counter2`, for storing the reference to the object that is to be created (see Figure 3.10).

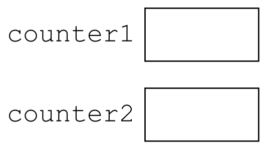


Figure 3.10 Two references for ticket counter objects

2 A new ticket counter is created and the reference is stored by the variable `counter1` (see Figure 3.11).

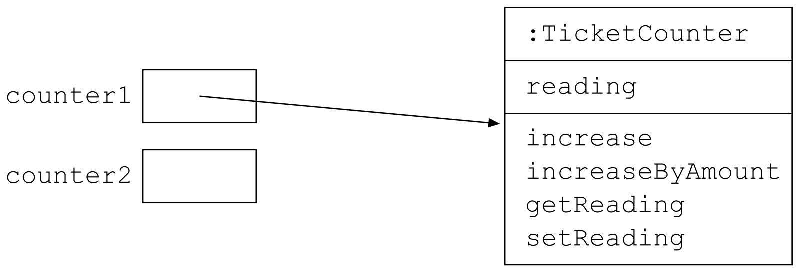


Figure 3.11 A ticket counter object is referenced by `counter1`

3 As we discussed in the previous section, the functionality of an equals sign is to *assign* the value of the expression on its right-hand side to the variable on its left-hand side. Therefore, the content of `counter1` is copied to `counter2` so that the contents are identical. Since the content of variable `counter1` is a reference to the ticket-counter object, the variable `counter2` also contains the same reference now (see Figure 3.12).

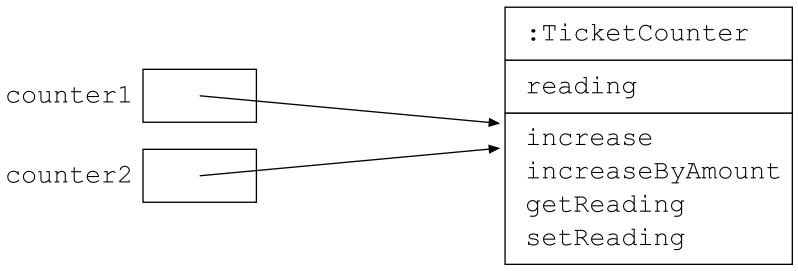


Figure 3.12 Both `counter1` and `counter2` are referencing the same ticket counter object

The main reason why it is necessary to store the reference by using more than one variable is that different variables that refer to the same object are used under different situations. We will encounter such situation often in the coming units.

In our real-life example, imagine that you find the ticket counter first by reading the message in the box labeled 'Ticket Counter 1' and set its count to 10. Some days later, you read the message in the box labeled 'Ticket Counter 2'. As the way to get the ticket counter is the same, you find the same ticket counter and you set its count to 11.

Can you imagine what the count will be when you find the ticket counter by reading the instruction in the 'Ticket Counter 1' box afterward? The answer is surely 11 instead of 10, because there has only been a single ticket counter right from the start.

Then, can you also figure out what the value of instance variable `reading` of the variable `counter1` (i.e., `counter1.reading`) will be after the following two statements are executed?

```
counter1.setReading(10);
counter2.setReading(11);
```

The value is 11. If you got the answer correct, congratulations! Please go on to the following self-test to make sure that you really understand. Otherwise, please take a rest and review this section again.

Self-test 3.4

Please study the following program segment:

```
TicketCounter counter1, counter2;
counter1 = new TicketCounter();
counter2 = new TicketCounter();
counter1.setReading(1);
counter2.setReading(2);
counter2 = counter1;
counter2.setReading(3);
```

Use the little box model or otherwise to answer the following questions:

1 How many ticket-counter objects are created?

2 How many ticket-counter objects are left at the end?

3 What is the content of the instance variable `reading` of the object referred to by `counter1`, i.e., `counter1.reading`?

Up to this point, we have only covered attributes, that is, instance variables. What we can do with variables is just read their contents or modify their contents by assigning new values to them. We now discuss how to manipulate the behaviours of classes, which are the methods.

Message sending among objects

In our `TicketCounter` class, there are four methods — `increase`, `increaseByAmount`, `getReading` and `setReading`. If you want to increase the ticket-counter object by one, which is similar to accessing the attributes of the objects, you have to get the ticket counter first and make it increase its count value by itself.

In real life, there must be a trigger or button on a ticket counter that you can press to increase the count value by one.

When you run your program, it is as if you send a message to the ticket-counter object so that it increases its value by itself. In programming, you use a statement like:

```
counter.increase();
```

To illustrate the above statement with the little box model, it's as if you use the little box labeled 'counter' to get the instruction to find the ticket-counter object. The parentheses that follow the word `increase` mean it is not an attribute but a behaviour. You therefore press the button on the ticket counter to pass a message `increase` to it in order to increase its value by one. At the time when the ticket counter intends to increase its value, it finds its own value, say 10, and then changes it to 11.

Back to our program example! The following (Figure 3.13) is the scenario before executing the above statement:

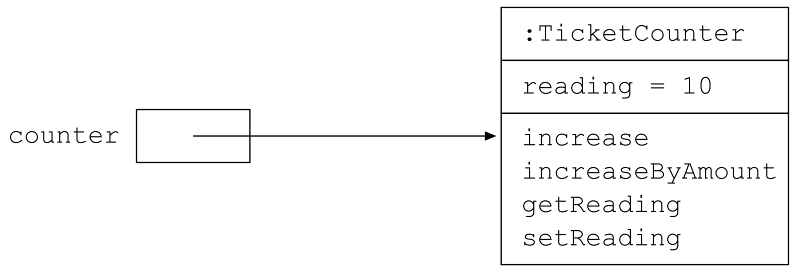


Figure 3.13 The initial state with `reading` being 10

The statement `counter.increase()` means sending a message (increase) to the object that the variable `counter` refers to so that the behaviour of the ticket-counter object is performed.

According to the discussion of the method `increase` earlier in this unit, you can see that the original value of the instance variable `reading` is used to calculate the expression

```
reading + 1
```

and the result is assigned to the instance variable `reading,` which practically increases the variable `reading` by one.

Therefore, after the statement `counter.increase()` is executed, the value of the instance variable `reading` becomes 11 (see Figure 3.14).

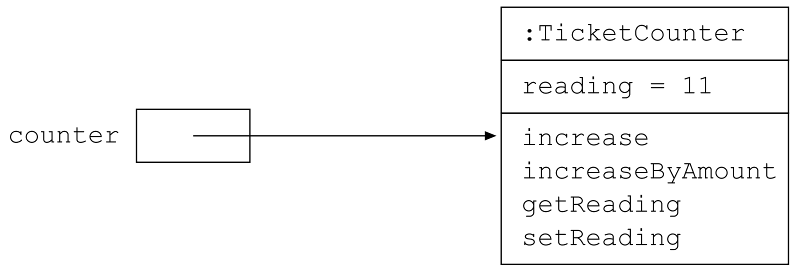


Figure 3.14 The state after an `increase` message has been sent

When you send a message to an object, you can provide extra data so that the object can perform its behaviour accordingly. For example, if you want to increase the count of a ticket-counter object by three at a time instead of one, you can send an `increaseByAmount` message [with an amount of three (3)] instead of sending three `increase` messages to the object.

To provide extra data in a message to an object, you provide parameters with the message in the parentheses. For example,

```
counter.increaseByAmount(3);
```

When the object performs its behaviour `increaseByAmount`, it gets an extra datum of three (3). Therefore, it increases its value by three. Since method `increaseByAmount()` is declared so that one parameter of type `int` is needed, you need to provide exactly one extra datum of `int` type when you send a message `increaseByAmount` to the object. Therefore, you should not have the following statements in your program:

```
counter.increaseByAmount();
counter.increaseByAmount(1, 2);
```

Let’s consider the following scenario. You are a staff member responsible for counting the attendees in a meeting. At the time you arrive at the venue, there are already 10 attendees. Afterwards, another attendee arrives, who is then followed by a group of three.

To model the above scenario, you need the following statements:

```
TicketCounter counter = new TicketCounter();
counter.setReading(10);
counter.increase();
counter.increaseByAmount(3);
```

Whenever you want to know the current reading of the counter, you can check the `reading` attribute of the `counter` object. If you want your counter to print the result on a slip, you may need to press the ‘print’ button on the counter to get the slip. To achieve this result, we have a `getReading` behaviour in the `TicketCounter` class. You can send a message `getReading` to the ticket counter when you run your program so that the object will give you (reply with) the result. The statement is:

```
counter.getReading();
```

Usually, you need to store the reply so that you can further manipulate it:

```
int theReading;
theReading = counter.getReading();
```

The first statement declares a variable `theReading` of `int` type so that it can store any integer value. For the second statement, the statement `counter.getReading()` sends a message `getReading` to the ticket-counter object and it replies with the current value. The equals sign stores the reply value into the variable on the left-hand side, i.e. the `theReading` variable.

If you want to count the tickets of female and male attendees, you need two ticket counters. In your program, you need to create two ticket counter objects, such as:

```
TicketCounter femaleCounter, maleCounter;
femaleCounter = new TicketCounter();
maleCounter = new TicketCounter();
```

Then every time a female attendee arrives, you press the ticket counter for female. That is, you send a message `increase` to the ticket counter for females. In a program, you would need the following statement:

```
femaleCounter.increase();
```

Similarly, the way to increase the ticket counter for males is:

```
maleCounter.increase();
```

The scenario can be visualized as in Figure 3.15:

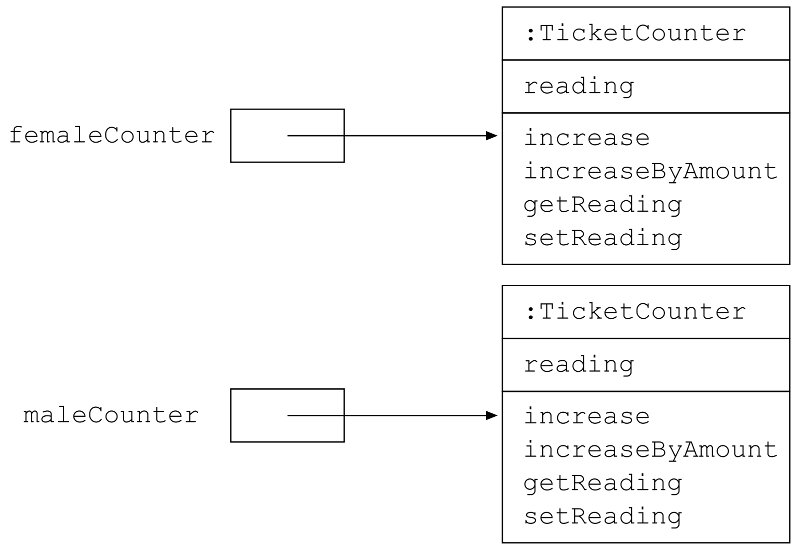


Figure 3.15 The variables `femaleCounter` and `maleCounter` are referring to two different ticket counter objects

Just before the meeting starts, you can determine the total number of attendees by getting the count values from the counters for females and males. In a program, you can use the following statement:

```
int total = femaleCounter.getReading() +
            maleCounter.getReading();
```

The sum of the `reading` values from the two counters is assigned to a variable `total`.

Self-test 3.5

Please study the following program segment.

```
TicketCounter counter1, counter2;
counter1 = new TicketCounter();
counter2 = counter1;
counter1.setReading(10);
counter1.increase();
counter2.increaseByAmount(2);
```

Can you use the little boxes as an analogy to explain what is going on in the above program segment?

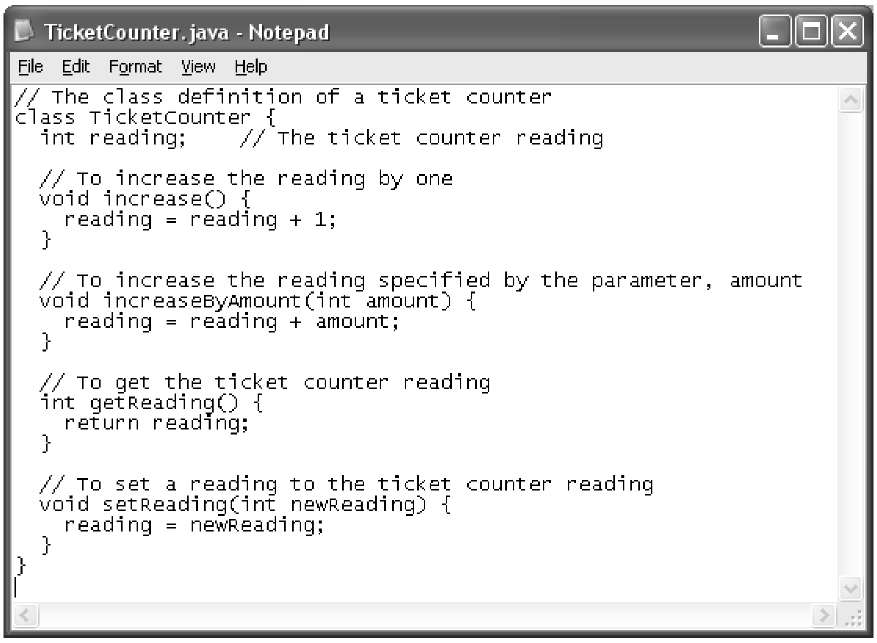
After the above statements are executed, what is the value of `reading` of the ticket-counter object?

Everything is now ready for you to try to write a complete Java program and have it executed by the computer.

First of all, you need to prepare a file that contains the programs. This file is called a *source code file*, or simply a *source file*. You can use an editor such as the Notepad, which comes with Microsoft Windows, to compose it. Please follow the steps below to prepare the definition of the `TicketCounter` class.

1 Start the Notepad application.

2 Enter the program listing as in Figure 3.16.



```
// The class definition of a ticket counter
class TicketCounter {
  int reading;    // The ticket counter reading

  // To increase the reading by one
  void increase() {
    reading = reading + 1;
  }

  // To increase the reading specified by the parameter, amount
  void increaseByAmount(int amount) {
    reading = reading + amount;
  }

  // To get the ticket counter reading
  int getReading() {
    return reading;
  }

  // To set a reading to the ticket counter reading
  void setReading(int newReading) {
    reading = newReading;
  }
}
```

Figure 3.16 TicketCounter.java

Save the file as `TicketCounter.java`

As a rule of thumb, a class with a name, say `MyClass`, must be defined in a source file named `MyClass.java`. Otherwise, your computer cannot compile and execute your program.

You need another program to model the real-world scenario by preparing an initial environment. In our case, we can use a program named `TestCounter` to do so. The following is a template that you can use to set up a model for a real-world scenario.

```
public class Program-name {
  public static void main(String args[]) {

  }
}
```

Your program can be placed after the second line. You can provide any valid identifier as the program name in the above template. For example, to model the real-world scenario in the previous self-test, you can have the following program listing:

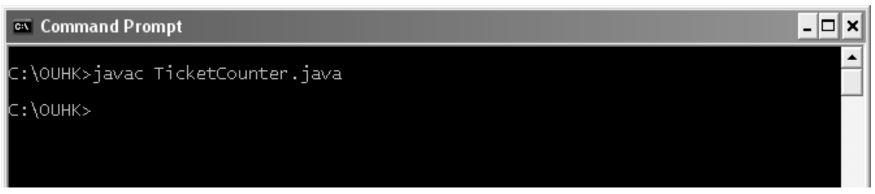
```
// The class to model the initial scenario
public class TestCounter {

  // The starting method to be executed
  public static void main(String args[]) {
    TicketCounter count1, count2;
    count1 = new TicketCounter();
    count2 = count1;
    count1.setReading(10);
    count1.increase();
    count2.increaseByAmount(2);
  }
}
```

For the above `TestCounter` program, you can follow the steps used in creating the `TicketCounter` source file to create this `TestCounter` source file. Please be careful that you save this source file as `TestCounter.java`.

You can now compile your source files. You need to start an MS-DOS Prompt (or Command Prompt) in Microsoft Windows, to carry out the task. Please enter

```
javac TestCounter.java
```

at the MS-DOS Prompt.



```
C:\OUHK>javac TicketCounter.java

C:\OUHK>
```

If you get any error message when you compile the program, please review *Unit 1*.

The compiler is smart enough to know that your program uses the `TicketCounter` class, and the `TicketCounter.java` source file is compiled as well.

To execute your program, simply enter

```
java TestCounter
```

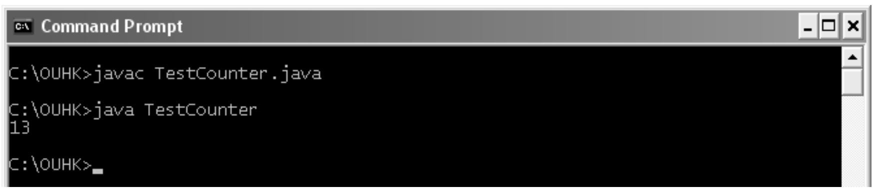
at the MS-DOS Prompt. You will find that your computer seems suspended for a while and nothing is shown. Nevertheless, your program has already done something. To give some output, let's modify the `TestCounter` source code by appending a statement

```
System.out.println(counter1.getReading());
```

and your program becomes:

```
public class TestCounter {
  public static void main(String args[]) {
    TicketCounter counter1, counter2;
    counter1 = new TicketCounter();
    counter2 = counter1;
    counter1.setReading(10);
    counter1.increase();
    counter2.increaseByAmount(2);
    System.out.println(counter1.getReading());
  }
}
```

The functionality of the statement `System.out.println(…)` is to send a message `println` to the system standard output device, which is the MS-DOS prompt in our case, containing the value of the variable (attribute) `reading` of the object referred to by `counter1`. When the standard output device gets the message, the value of the variable `reading` of the object is displayed:

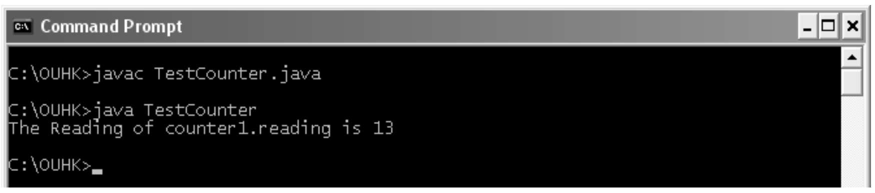


```
C:\OUHK>javac TestCounter.java

C:\OUHK>java TestCounter
13

C:\OUHK>_
```

If you want the output message to be more informative, you may add a prefix to the value:

```
System.out.println("The Reading of counter1.reading is " +
                   counter1.getReading());
```

Re-compile the program and you'll get:



```
C:\OUHK>javac TestCounter.java

C:\OUHK>java TestCounter
The Reading of counter1.reading is 13

C:\OUHK>_
```

Access control — private and public

In the ticket-counter example, we can access the attribute `reading` either directly or by sending a message to the object to access the value indirectly. For example, the following two program segments

```
TicketCounter counter = new TicketCounter();
counter.reading = 1;
```

and

```
TicketCounter counter = new TicketCounter();
counter.setReading(1);
```

are practically equivalent.

We have said that the latter is preferable but the former seems to be simpler. Therefore, you might be tempted to use the former when you are writing your Java programs.

You can use access control to limit the object attributes and methods that can be accessed by other objects when you run your program.

To prevent direct access to an object attribute, you can add a modifier `private` to the corresponding variable. It is a good programming practice and you should stick to it. You will sooner or later find that your programs are easier to maintain and are more reliable.

For example, the definition of the class `TicketCounter` can be modified to be:

```
// The class definition of a ticket counter
class TicketCounter {
  private int reading;     // The ticket counter reading

  // To increase the reading by one
  public void increase() {
    reading= reading + 1;
  }

  // To increase the reading specified by the parameter, amount
  public void increaseByAmount(int amount) {
    reading= reading + amount;
  }

  // To get the ticket counter reading
  public int getReading() {
    return reading;
  }
  // To set a value to the ticket counter reading
  public void setReading(int newReading) {
    reading = newReading;
  }
}
```

In the above class definition, the instance variable `reading` is marked as `private`, which means that this instance variable `reading` can only be accessed by a `TicketCounter` object. Therefore, if you have the following program section

```
TicketCounter counter = new TicketCounter();
counter.reading = 1;
```

when you compile the program, you will be prompted with an error message.

All instance methods above are marked as `public`. This makes these methods publicly accessible by all other objects. Then, the only way to change the ticket counter's attribute `reading` is to send messages to the counter, which changes its attribute `reading` by itself.

What can Java programs manipulate?

Unit 1 states that a computer is a programmable device that can manipulate data based on some predefined instructions, which are the programs. Now that you know some fundamental concepts for writing Java programs, we can discuss what data your Java program can manipulate.

In Java, there are only two kinds of data types, *primitive* types and *non-primitive* types. We discuss them in detail below.

Primitive data types

The word *primitive* means basic or fundamental. Therefore, primitive data types are the most fundamental data types, such as numeric data types. There are eight primitive types in total in Java. They are `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. Any types other than these eight are known as *non-primitive* types.

You have seen how to declare variables of primitive types in the previous sections. You can declare a variable of any type in the following way:

variable-type variable-name;

If you declare a primitive type variable, it is as if you have a little box that can store a piece of paper that has a value on it. For example, you declare a variable as follows:

```
int temperature;
```

The above statement is analogous to preparing a little box with a label ‘temperature’ on it and you can place a piece of paper with a value written on it in the box.

In the previous section, you encountered some declarations such as:

```
TicketCounter counter;
```

The type is `TicketCounter`, which means that it is a *non-primitive* type. If we use the little boxes to illustrate the differences, the main difference is that the box for a primitive type (the temperature variable) stores the *value,* whereas the box for a non-primitive type (the counter variable) stores the *way* to find or get the real object. When you write your program, a ticket counter can be created using the following two statements:

```
TicketCounter counter;
counter = new TicketCounter();
```

Compared with the declaration of a non-primitive variable, the declaration of a primitive variable means that it can be used immediately, such as:

```
int temperature;
temperature = 20;
```

You can always use a shorthand form to combine the above two groups of statements:

```
TicketCounter counter = new TicketCounter();
int temperature = 20;
```

Integral types

We mentioned that the values of some attributes have to be in whole numbers or integers (numbers without fractions) and that you can use a type named `int` to denote an integral type. There are, however, four integral types: `byte`, `short`, `int` and `long`. They can only store whole numbers (integers). The differences among them are that the computer uses different memory sizes to store the variables of these four integral types, and thus the range of values that can be represented is also different. The following table shows their sizes and ranges.

Data type	**Memory size**	**Range**
`byte`	8 bits	-128 to +127
`short`	16 bits	-32768 to +32767
`int`	32 bits	-2147483648 to +2147483647
`long`	64 bits	-9223372036854775808 to +9223372036854775807

A memory size of 8 bits means a sequence of eight binary digits, values either 0 or 1. There are 2^8 (i.e. 256) combinations in total. All Java integral types can represent both positive and negative integers. Half of the combinations represent negative values (i.e. -1 to -128) and half of them represent non-negative values (0 to 127), so the range is from –128 to +127. You can similarly determine the range of the variables of the `short`, `int` and `long` types.

For example, if you have an `aByte` variable of type `byte`, which is declared below as

```
byte aByte;
```

a memory block of 8 bits is reserved in the computer and is denoted by the name `aByte`. You can assign any integer within the range of -128 to +127 inclusive to it. If you think the range is not large enough, you can use the type `short`, `int` or even `long`.

For simplicity, you can use `int` as the type for variables that store integral values because its range is sufficiently large for common usage.

If you want to represent an integral literal (a constant) in your program, you can simply write down the number. A numeric literal without a decimal point in your program is by default integral. Examples are 2 and -1.

You can also specify the memory size to store these literals. (The computer also needs some memory space to store such literals.) If you just write down a numeric literal without a decimal point, it is stored in the memory using 32 bits by default, which is equivalent to `int` type variable storage. The Java programming language provides the flexibility to store an integral value using 64 bits. You can add a suffix of letter L or l to the numeric literals, such as 10L or 5l. As the small letter l cannot be easily recognized when reading the program listing [you might think it is a 1 (one)], you are advised to use the capital letter L.

If you think specifying an integral value in octal or hexadecimal can make your program more readable, you can use a prefix 0 for octal numeric values and 0x or 0X for hexadecimal numeric values. For example, 077 in your program means 77_8 and 0xabcd (or 0xABCD) represents a value of $ABCD_{16}$ as in the following statements:

```
int var1 = 077;
int var2 = 0xABCD;
```

Real types

For processing numeric data with fractions, you need to use another category of data type, which is usually known as *real* or *floating-point* type. The name ‘floating-point’ reflects the fact that the decimal point can be freely placed between any two digits in a number; for example, you can use real type to represent values 1.23 and 12.3.

The Java programming language provides two different *real* types. The differences between them are mainly in the memory size required to represent a value and thus in the range and precision of the value that can be represented.

Data type	**Memory size**	**Range**
float	32 bits	Maximum value: 3.4028235×10^{38} Minimum value: 1.4×10^{-45}
double	64 bits	Maximum value: $1.7976931348623157\times10^{308}$ Minimum value: 4.9×10^{-324}

Both of the above data types can represent positive and negative values. The maximum values are the ones with the largest magnitudes, which can either be positive or negative. The minimum values are the ones that are closest to zero but are still not perceived as zero.

To express a *real* literal in your Java program, you can simply write down a numeric value with a decimal point, such as 3.14. By default, the Java program uses 64 bits to represent a floating-point literal, which is equivalent to a `double` type. Therefore, 3.14 is represented by 64 bits in the computer. To restrict the computer to use only 32 bits for storing a real value, you can add suffix `F` or `f` (`float`) to a real literal, for example, 3.1415F or 3.1415f.

The following examples declare variables of `float` and `double` respectively with explicit initialization:

```
float pi = 3.14F;
double e = 2.71828;
```

For scientific calculations, you might sometimes find it useful to express a value in scientific notation. For example, you might use `5.234E20` to denote a value of 5.234×10^{20} in your program. The numbers that follow the letter `E` are presumed to be positive. To represent a negative exponential value, use a negative sign immediately after the letter `E`. For example, the minimum positive value in the Java programming language is 1.4E-45. Scientific notation helps you visualize the numeric values, especially the magnitudes. No matter how a number is expressed, it is represented the same in the computer.

Boolean type

A `boolean` type variable can store only two possible values — `true` and `false`.

For example, a variable `isValid` that can either store `true` or `false` is expressed in the following way:

```
boolean isValid;
```

There are two pre-defined literals, `true` and `false` in the Java programming language. Therefore, you can assign a value of either `true` or `false` to a `boolean` variable type. For example,

```
isValid = true;
```

You can also use shorthand here so that the above two statements become

```
boolean isValid = true;
```

Character type

Besides manipulating numeric data, another category of data that you will probably manipulate is textual. For example, the size of a T-shirt can be ‘S’, ‘M’ or ‘L’. Then, you need to use a type that can store any digit, letter, or a special character. The Java programming language provides a `char` type that can store such data.

To declare a variable `size` that can store a letter, you can use the following statement:

```
char size;
```

In Java, all characters are represented in 16-bit Unicode, which is an encoding system that embeds many existing coding systems used by different countries. It is possible to use a character to represent a single-byte English alphabet (letter) or a double-byte Chinese character. Therefore, Java is commonly used to develop applications with internationalization in mind.

You use a pair of *single* quotation marks to specify a character literal. For example,

```
size = 'S';
```

If you have the Unicode number of a character you want to specify in your program, you can specify it as:

```
specialChar = '\uABCD';
```

The format is a pair of single quotation marks to quote the Unicode number in hexadecimal with prefix `\u`. Therefore, `'\uABCD'` represents a single character with Unicode number $ABCD_{16}$.

There are also some special characters pre-defined in Java. They are:

`'\n'` for the new line character
`'\r'` for the carriage return (<enter>) character
`'\t'` for the tab character

The `char` type can store only one character. If you want to specify a text string, you need to use `String` objects. In Java, you use a pair of *double* quotation marks to specify a text string. For example,

```
"MT201"
```

If you write down the above string literal in your program, it represents a string object when you run your program. Therefore, you need a variable of `String` type to refer to it. You could use the following statement to do it:

```
String course = "MT201";
```

As you can see that the type `String` is not one of the primitive types, it is therefore a non-primitive type that has its set of attributes and behaviours. The `String` class is discussed in detail in subsequent units of *MT201*.

Classes

Classes or non-primitive types are types other than the eight primitive ones. As we mentioned in the previous section, non-primitive types can be used to model objects that have both attributes and behaviours.

Comparisons of primitive data types and non-primitive types

Declaring a primitive type variable in Java programs allocates a memory location, and it can store any acceptable value of the corresponding primitive type. As a primitive variable refers to a datum that has no behaviour and no attribute, you will never use dot-notation with a primitive variable. The primitive types are pre-defined, and you cannot build and use your own primitive types.

Variables of non-primitive types, you might recall, are also known as *reference* types. Declaring a non-primitive type variable allocates a memory block that just stores the handle (or reference) to an object in the memory, but no object is created. If you need objects when you run your programs, you need to create them and you usually store their references in non-primitive variables of compatible types.

Suppose that we have defined a class named `Square`. Let’s consider the following two statements:

```
int i; Square x;
i = 123; x = new Square();
i = 456; x = new Square();
```

Each line in the program segment above contains two statements for clearer comparison, but it is a good programming practice to place only one statement on each line.

For the first line, two variables `i` and `x` of type `int` and `Square` respectively are defined. The statements in the second line assign the value 123 to variable `i` and the reference to a created `Square` object to variable `x`. The following diagram (Figure 3.17) should help you visualize the situation:

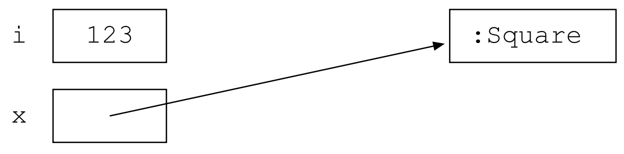


Figure 3.17 Visualizing the state just after executing the statements of the second line

You can see that the variable `i` stores the numeric value 123 and the variable `x` stores the reference to the `Square` object.

The third line of the program segment assigns a numeric value 456 to variable `i`, creates a new `Square` object and assigns the object reference to the variable `x`. The scenario becomes (see Figure 3.18):

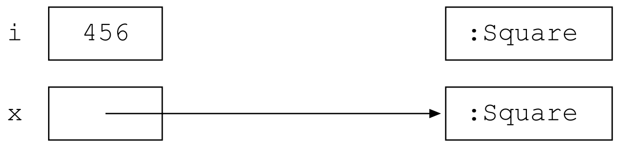


Figure 3.18 The state just after executing the statements of the third line

How about the following program segment?

```
int i, j; Square x, y;
i = 123; x = new Square();
j = i; y = x;
j = 456; y = new Square();
```

Can you figure out the operations to be carried out and the situation after the above four lines of statements are executed?

Like the previous program segment, the first line declares variables `i` and `j` of type `int` and variables `x` and `y` of type `Square`. The second line assigns numeric value 123 and the reference to a `Square` object to variables `i` and `x` respectively. The situation becomes (see Figure 3.19):

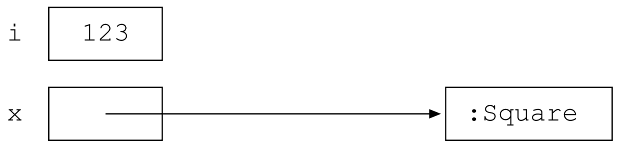


Figure 3.19 Visualizing the state just after executing the statements of the second line

The third line assigns the contents of variables `i` and `x` to the variables `j` and `y` respectively. The situation becomes (see Figure 3.20):

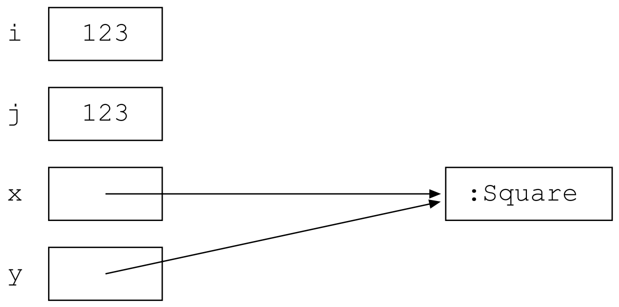


Figure 3.20 The state just after executing the statements of the third line

In the fourth line, the statement ‘`j = 456;`’ assigns a numeric value 456 to the variable `j`. Do you think the statement will change the content of variable `i` as well? The answer is definitely not. Therefore, the circumstance for the primitive variables becomes (see Figure 3.21):

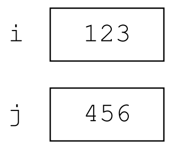


Figure 3.21 The variables `i` and `j` just after executing the first statement of the fourth line

Similarly, the statement '`y = new Square();`' creates a new `Square` object and assigns the object reference to the variable `y`. The contents of variable `x` will never change. Then, the scenario for non-primitive type variables `x` and `y` can be seen in Figure 3.22:

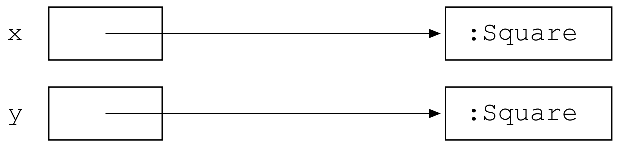


Figure 3.22 The ticket counter objects just after executing the second statement of the fourth line

You can see that there are two `Square` objects that are referenced by variables `x` and `y` respectively.

There is no limit to the number of classes, and it is up to you to design your own classes whenever you think they are necessary.

Choosing suitable attribute types

You now know that there are eight primitive types and that any other types are known as non-primitive types. When you have analysed your problem and derived sets of object attributes and behaviours, the next step is to carefully select a suitable type for each attribute. The selection process not only helps you in understanding the objects in the problem better, but it also minimizes the need for changing variable types when you develop the software.

First of all, you prepare a list of object attributes. Here is an attribute list for a bank account object as an example:

1 Account owner's name

2 Owner ID number

3 Gender

4 Address

5 Day of birth

6 Month of birth

7 Year of birth

8 Day of account opening

9 Month of account opening

10 Year of account opening

11 Account number

12 Account type

13 Interest rate

14 Account balance

15 Active Account

First of all, you should classify all attributes into different primitive type categories, such as numeric, textual and logical.

For numeric types, the attribute value is usually obviously numeric and its contents might involve numeric calculations. Furthermore, you are usually concerned with its value rather than the format, i.e. the way to present it. Therefore, leading zeroes can usually be ignored. From the above attribute list, the following attributes should be classified as numeric type: account balance, interest rate, day of birth, month of birth, year of birth, day of account opening, month of account opening, year of account opening.

An attribute that is classified as a textual type can include non-digit characters. Its values are usually not used in numerical calculations but in pattern matching. The format for storing it is also a consideration. Numbers with leading zeros are considered as a textual type. From the above attribute list, account name, owner identity number, account type, account number, gender and address are commonly considered as a textual type.

You might wonder why both owner identity number and account number are classified as textual types even though the attribute names imply they are numbers. In Hong Kong, the identity number contains at least one leading letter, and only a textual type can store this leading letter of the alphabet. An account number is usually not involved in any numeric calculation but is only for matching. Furthermore, its format, which often involves leading zeros, makes it practically into a textual type.

If an attribute can contain only two values, such as positive/negative, yes/no, true/false, it is classified as a logical type. The attribute active account in the above list can be considered a logical type. In the program, you can assign the primitive type `boolean` to it.

You have to classify the numeric types and textual types further.

For a textual type, if an attribute value can only be a single character, you can assign `char` type to that attribute. Otherwise, it should be considered a `String` type. For example, in the above list of textual attributes, account type (`'S'` for savings account and `'C'` for current account) and gender (`'F'` for female and `'M'` for male) can be `char` — the others are all `String` types. Therefore,

char: account type, gender
String: account name, owner identity number, account number and address

Can we use a numeric type to represent account type and gender type? The answer is surely yes. For example, for the gender attribute, you can use 1 to represent male and 2 to represent female. However, whenever you process the gender attribute, you need to remember the meanings of 1 and 2. The use of `'M'` and `'F'` is self-explanatory and reduces the complexity when developing the program.

There are two further steps to classify numeric data. The first step is to determine whether an attribute value must contain fractional parts. Such attributes should be further classified as *real* type; otherwise, they should be classified as *integer* type. Therefore, the above numeric type list can further be subdivided as:

integer: day of birth, month of birth, year of birth, day of account opening, month of account opening, year of account opening
real: account balance, interest rate.

A variable of `float` or `double` type can store a value with a fraction. They, of course, can also store integral values, which means that the numeric values have fractions of all zeros. Then, why don’t we simply declare all numeric types as *real* types? The reason is: Real numbers are stored in computers by approximation, whereas integral numbers are stored exactly. Therefore, whenever it is possible to use an integer type, you should use it so that you do not face the risk of losing precision.

The next step in classifying numeric types is to investigate the possible range of the attributes, so that you can choose a suitable type for them. For example, the possible value of day of birth and day of account open is 1 to 31. Therefore, the primitive type `byte` is sufficient. Furthermore, primitive type `float` is large enough to store any possible value of account balance and interest rate.

However, I would recommend that you use `int` for all integral attributes and `double` for attributes whose value can contain fractions. The reason is that `int` and `double` are the default types for integral literals and real literals. If you use `int` and `double` only, you can bypass many troublesome restrictions in the Java programming language, such as the type conversions that would take place during numerical calculations.

As a result, the numeric attribute lists become:

`int`: day of birth, month of birth, year of birth, day of account opening, month of account opening, year of account opening
`double`: account balance, interest rate

Therefore, you can determine the class definition (without behaviour) of a bank account as:

```
// The class definition of a bank account
class BankAccount {
  String name;             // Owner name
  String idNumber;         // Owner ID number
  String accountNumber;    // Account name
  char type;               // Account type
  char gender;             // Owner gender
  double balance;          // Account balance in HK dollars
  double interestRate;     // Account interest rate
  String address;          // Owner address
  int dayOfBirth;          // Owner’s day of birth
  int monthOfBirth;        // Owner’s month of birth
  int yearOfBirth;         // Owner’s year of birth
  int dayOfAccountOpen;    // Owner’s day of opening A/C
  int monthOfAccountOpen;  // Owner’s month of opening A/C
  int yearOfAccountOpen;   // Owner’s year of opening A/C
  boolean isActiveAccount; // A dormant account?
}
```

The above is *one* possible way to define the bank account class — you might do it differently.

You can have a further step in deriving the attribute types. In the above example, you can see that the three attributes — day, month and year — are actually co-related to one another. Furthermore, there are two sets of day, month and year in the attribute list. Therefore, you can further consolidate the attribute lists by making parts of the list another non-primitive type.

For example, you can derive a class definition that only contains day, month and year. That is,

```
// The class definition of a date
class Date {
  int day;
  int month;
  int year;
}
```

You can modify the bank account class definition further by grouping the two sets of day, month and year. Then, the class definition becomes:

```
// The class definition of a bank account
class BankAccount {
  String name;              // Owner name
  String idNumber;          // Owner ID number
  String accountNumber;     // Account name
  char type;                // Account type
  char gender;              // Owner gender
  double balance;           // Account balance in HK dollars
  double interestRate;      // Account interest rate
  String address;           // Owner address
  Date dateOfBirth;         // Owner’s date of birth
  Date dateOfAccountOpen;   // Owner’s date of opening A/C
  boolean isActiveAccount;  // A dormant account?
}
```

Notice that the type of `dateOfBirth` and `dateOfAccountOpen` is `Date`, which is not a primitive type. Therefore, the variables only store a reference to a `Date` object. As a result, it is necessary to create two `Date` objects for date of birth and date of account opening. You might comment that using the class type, `Date`, does not benefit you in any way but only makes the program more complicated. However, using a class `Date` enables you to develop it further and make it complete. Then, you can reuse this class definition in any problem you encounter in the future. The details of creation of `Date` objects and initialization of the `BankAccount` class are covered in *Unit 7*.

In conclusion, there is usually more than one possible classification, and the result is determined by your understanding of the object (and hence the attributes) as well as your experience.

Manipulating data

A computer is actually nothing more than a programmable calculation device. Therefore, you need to know how to manipulate data such as performing basic numeric calculations by writing Java programs.

In the following sections, we discuss the way to perform numeric calculations. Boolean is a special type in that Boolean values cannot mix with other numeric types.

Assignment

In the previous section, you encountered the use of the equals sign. Once again, the equals sign is not for comparing the values on both sides but to store the value of the right-hand side expression to the variable on the left-hand side. This is known as an *assignment* operation. Therefore, there is always a *variable* on the left-hand side of an assignment operation, and you will never have statements like

```
a + 1 = b + 2;
```

because the left-hand side is not a variable and it cannot store the result of the expression on the right-hand side.

Numerical calculations

You can perform numerical calculations in your program. Up to now, you have encountered simple addition operations using the plus sign (+). For example:

```
int a = 1;
int b = a + 2;
```

The second statement performs an addition operation to obtain the result of the content of variable `a` plus 2. Therefore, after the above statements are executed, the value of variable `b` is 3.

Similarly, you can perform a subtraction in your program using the minus sign (-). For example:

```
int a = 3;
int b = a – 1;
```

The expression on the right-hand side of the equals sign is obtained first, which is 3 – 1 or 2, and the result 2 is assigned to variable `b`.

You can perform two other numeric calculations, multiplication and division, using the * and / operators. You should be warned that × and ÷ are not valid operators in the Java programming language.

Let’s consider the following statements:

```
int a = 5, b = 2;
int c = a / b;
```

It is obvious that 5 ÷ 2 in a usual mathematical calculation is 2.5. However, integer division in Java programming gives an *integer* result only. Therefore, the result of the above division is 2. The fractional part of the division result is *truncated*, which means the fractional part is ‘thrown away’ from the division result.

Operator precedence

An expression in your program can contain more than one operator. For example,

```
int a = 1, b = 2, c = 3;
int d = a + b * c;
```

Your computer is clever enough to perform multiplication and division first. Addition and subtraction are performed afterwards. Therefore, the above expression is equivalent to:

= 1 + (2 × 3)
= 1 + 6
= 7

A value of 7 is assigned to variable `d`.

The precedence of the four operators is:

```
* /   |
+ -   ↓
```

That is, multiplication and division are always performed before addition and subtraction.

If there is more than one operator at the same level, the operators are performed from left to right. For example,

```
int a = 1, b = 2, c = 3, d = 4;
int e = a + b * c / d;
```

Multiplication and division are performed first. As * and / are at the same level, the one on the left is performed first. The above expression is equivalent to:

= 1 + 2 * 3 / 4
= 1 + 6 / 4
= 1 + 1
= 2

You should notice that 6/4 equals 1 in integer division. Let’s consider a similar statement:

```
double a =1.0, b = 2.0, c = 3.0, d = 4.0;
double e = a + b * c / d;
```

The result is determined as:

= 1.0 + 2.0 * 3.0 / 4.0
= 1.0 + 6.0 / 4.0
= 1.0 + 1.5
= 2.5

The overall result is 2.5.

If you want to alter the order in which the operators are performed, you can use parentheses. For example:

```
int a = 1, b = 2, c = 3;
int d = (a + b) * c;
```

The expression in the parentheses is performed first. Therefore, the above expression is evaluated as:

= (1 + 2) * 3
= 3 * 3
= 9

If necessary, you can even nest parentheses in the expression. For example, you can have an expression like:

```
int a = 1, b = 2, c = 3, d=4;
int e = (a – (b + c)) * d;
```

The expression is evaluated as:

= (1 – (2 + 3)) * 4
= (1 – 5) * 4
= -4 * 4
= -16

Self-test 3.6

Given

```
int a = 1, b = 2, c = 3, d = 4;
```

please determine the results of following expressions:

1 a – b + c

2 c * d / b

3 (a + c) * d / c

4 d * d / b / b

Implicit and explicit type conversions

Up to now, you have frequently seen statements like:

```
int a = 10;
```

We have mentioned that a numeric literal in your program is an `int` type by default, which is stored using 32-bits. What do you think about the following statement?

```
long b = 10;
```

First of all, the variable `b` is a `long` type that can store a 64-bit integer value. The value on the right-hand side is an `int` type by default, which only needs 32 bits. Is it possible to assign this 32-bit integer value to the 64-bit variable of a long type?

The answer is yes, because the computer knows that it has enough storage to store the value and it can just leave those extra 32 bits unused, or to be more exact the program assigns zeros to the unused bits. Such automatic conversion by the computer is known as *implicit conversion*.

Now, what do you think about the following statement?

```
int c = 10L;
```

We mentioned that a numeric literal with a suffix `L` means that it is an integer that needs 64 bits to store it. However, the left-hand side is a variable that can only store a 32-bit integer. Even though the value 10 is well within the possible range of an `int` type, the above statement is invalid.

Similarly, what do you think about the following two statements?

```
double d = 3.14F;
float f = 3.1415;
```

You should notice that a numeric literal with a decimal point is *by default* a `double` type, which needs 64 bits to represent it. As there is a suffix `F` in the literal 3.14F, it is a `float` type that needs 32 bits. Therefore, the first statement is perfectly acceptable but the second one is invalid — the second is a `double` type (64 bit) by default.

From the above discussion, you should note that whether a value can be stored in a variable is not determined by its actual value. The sole restriction is that if the variable on the left-hand side does not have sufficient storage (by comparing the number of bits), the assignment operation would fail.

Another scenario is about an assignment involving integer and real values. What do you think about the following statement?

```
float f = 100;
```

You can see that the above assignment is to store an integer value 100 to a variable `f` of `float` type. Do you think that it makes sense to do so?

In Java programming, you can assign an integer value to a real type variable, which is either a `float` or `double`, but it is not possible to do it the other way round. Let's consider the following statement:

```
int j = 1.5;
```

You can see that the variable on the left-hand side is an integer type. It cannot contain any fraction. If the computer accepted it, the variable could only store an integer value of 1 and the fraction will be lost. Therefore, the above statement is not acceptable. What about the following?

```
int k = 0.0;
```

Even though the expression on the right-hand side is a whole number in our understanding, the computer still treats it as a number with a fraction. Therefore, the above statement fails as well.

In the above discussion, assignment operations fail if the assignment loses its precision (such as the fractional part of a real number is lost) or variables do not have sufficient storage to store the value.

In some instances, if you tell the computer that it is your intention and you take the risk of the assignment, the computer will agree and will perform the assignment operation for you.

A variable on the left-hand side of an equals sign is known as *assignment compatible* if it is 'large' enough to store the value of the expression on the right-hand side. How can we compare the numeric primitive type to determine which one is larger?

First of all, please study the following diagram (Figure 3.23):

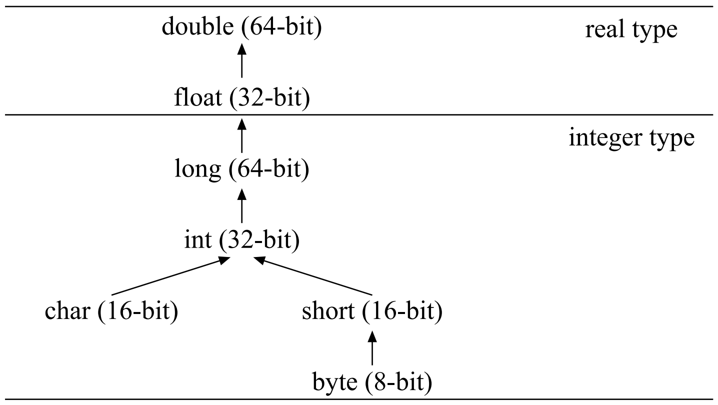


Figure 3.23 Conversion of some primitive types

If the type on the right-hand side of an equals sign is assigned to the variable with types that follow the arrows in the above diagram, the computer will automatically accept your assignment. For example, you can assign a `long` type value to a `float` type variable.

In the above diagram, you can see that the `char` type is also involved. In the Java programming language, the `char` type can also be considered an integer type. Both `char` and `short` are 16-bit data types and the difference is their ranges. The ranges of `char` and `short` are:

Data type	**Range**
char	0 to 65535
short	-32768 to 32767

Therefore, `char` can only store positive values, whereas `short` can store both negative and positive values.

A value of any data type in the above diagram can be assigned to a variable of another data type. If the assignment is not acceptable automatically, you need to tell the computer that you intentionally agree to the assignment even though the data precision or part of the data is lost. You can perform such *explicit conversion* by *casting*.

A general format for casting, which converts one type to another type, is:

(*variable-type*) *expression*

For example, if you want to assign a `double` value to an `int` variable, you can use the following statement:

```
int j = (int) 1.0;
```

The numeric literal 1.0 is a `double` type. If you want to assign it to an `int` variable, you precede the expression with parentheses that enclose the target type you want the computer to convert the value to, which is an `int` in the above example. After the statement is executed, the variable `j` stores a value of 1.

Consider the following statement:

```
int k = (int) 1.5;
```

The computer also knows that you want to convert a `double` type value to an `int` variable. As an `int` variable cannot contain a fraction, the fractional part of the value on the right-hand side of the equals sign is ignored or truncated. Therefore, the variable `k` stores a value of 1. You can see that some data (precision) is lost and the assignment would have failed if you had not used casting.

An exceptional case is the assignment that involves `char` and `short`. Both types are 16-bit integer values but their ranges are different.

Therefore, assigning a `char` type value to a `short` type variable or a `short` type value to a `char` type variable needs casting. For example,

```
char c = 'A';
short s = (short) c;
```

How about assigning a negative integer value to a `char` type variable? With casting, the computer will accept the assignment operation, and the 16 bits of the negative integer value are assigned to the `char` type variable. For example, if you have the following program segment,

```
short s = -1;
char c = (char) s;
```

the computer uses the following 16 bits to represent the value of –1:

```
1111111111111111
```

By the assignment operation, the sequence of the above 16 bits is copied to the `char` type variable `c`. As a `char` type variable can only represent positive values, value represented by the above sequence of 16-bit pattern is 65535. Therefore, when you display the content of variable `c`, the computer gives an output of 65535.

Self-test 3.7

Provided that there are variable declarations as follows,

```
double d;
float f;
long l;
int i;
short s;
char c;
byte b;
```

please determine whether the following assignments need casting:

1 `c = i;`

2 `f  = d;`

3 `i = b;`

4 `c = s;`

Class packages

You can now define Java classes to model real-world objects. Sooner or later, you will find that you have many class definitions and it becomes more and more difficult to identify the class definitions and their uses.

The necessities of class packages

For example, imagine that you are developing a payroll system that handles salary calculations. As different staff types and staff departments use different salary calculations, you define several class definitions for different staff types. For example:

```
class HRStaff { …… }
class EDPStaff { …… }
class AccountStaff { …… }
```

Furthermore, each department has its own benefit scheme. Therefore, you define other classes, such as:

```
class HRStaffScheme { …… }
class EDPStaffScheme { …… }
class AccountStaffScheme { …… }
```

You can imagine that if you have more departments, you will have a lot of classes and their names are lengthy, and it would be tedious to write them in your program. Therefore, there should be a more systematic way to organize the class names. Java provides a class package mechanism.

The package declaration

We can reorganize the class definitions above by using the class package mechanism. For example, you can change the `HRStaff` class to the following:

```
package hr;

class Staff {
……
}
```

The package declaration indicates the package to which the class definition belongs. In the program segment above, the `Staff` class belongs to the `hr` package. In each Java source file (i.e. the program file with file extension `.java`), there is at most one `package` declaration, and it must be the first statement in the source file. Then, the fully qualified name of the above staff class is:

```
hr.Staff
```

Similarly, you can add the package declaration to other classes so that you can have the following classes:

```
hr.Staff
hr.StaffScheme
edp.Staff
edp.StaffScheme
account.Staff
account.StaffScheme
```

The file names of the Java source files are the same as the class names. Therefore, the file names of the above `hr.Staff`, `edp.Staff` and `account.Staff` are all `Staff.java`. As a result, you will have to create subdirectories `hr`, `edp` and `account` yourself and store the Java source files in them accordingly.

If you want to create an object based on the class `account.Staff`, you need the following statement:

```
account.Staff staff = new account.Staff();
```

If your program refers to the `account.Staff` class frequently, writing such a long class name can be tedious.

The `import` declaration

To prevent your program from requiring lengthy class names, you can use the `import` declaration.

For example, you can place an import declaration in your source file as:

```
import account.Staff;
```

Then, anywhere in your program, wherever you refer to the class name, `Staff`, it is the `account.Staff.` For example,

```
import account.Staff;

public class TestStaff {
  public static void main(String args[]) {
    Staff s = new Staff();
    ……
  }
}
```

If you need to refer to both `hr.Staff` and `edp.Staff` in the same source file, you cannot simply refer to the class name, `Staff`, since the computer cannot identify which class you are actually referring to, and the compilation would therefore fail. Thus, you have to provide a fully qualified class name, which is the class name, and the package name as the prefix, in the program. For example,

```
public class TestStaff {
  public static void main(String args[]) {
    hr.Staff s1 = new hr.Staff();
    edp.Staff s2 = new edp.Staff();
    ……
  }
}
```

If your program refers to both `account.Staff` and `account.StaffScheme`, you can use two `import` statements,

```
import account.Staff;
import account.StaffScheme;
```

or one `import` statement like:

```
import account.*;
```

As a summary, the layout of a Java program must be:

package-declaration
import-declarations
class-definitions

Class members

In some cases, all objects of the same class share the same attribute. For example, all saving accounts in the same bank share the same interest rate. Therefore, instead of having an instance variable `interestRate` for each object, it is better to have a single attribute `interest` to be shared by all objects. Such shared attributes are known as *class variables*. Similarly, you can have class methods but they are methods for the class.

What are class members?

Up to now, all variables and methods defined in a class definition were known as *instance* variables and *instance* methods because they were associated with an object or instance. Instance methods operate on the instance variables of the same object. Each object has its own variables and methods, and objects of the same class by default share nothing.

Another problem with the instance variables and instance methods is that they are only available when you create the object. If you want to have attributes and methods that exist even before an object is created, you can use *class members*, including *class* variables and methods.

The `static` keyword

The keyword `static` is used to define class members. For the bank account example, we can have the following class definition:

```
// The class definition of an account
class Account {
  double interestRate;  // Interest rate
  double balance;       // Account balance
  ……
}
```

Then, every object you create with the above `Account` class has its own instance attributes `interestRate` and `balance`. If you want to have an attribute `interestRate` that could be shared by all `Account` objects, you can mark the attribute `interestRate` as `static` and the class definition becomes:

```
// The class definition of an account
class Account {
  static double interestRate;  // Common interest rate
  double balance;              // Account balance
  ……
}
```

With this new class definition, every object created does not have its own `interestRate` attribute but shares the same class variable `interestRate`. For example, there is a behaviour `calculateInterest` in the class `Account` and the class definition becomes:

```
// The class definition of an account
class Account {
  static double interestRate;  // Common interest rate
  double balance;              // Account balance

  // To calculate the interest
  double calculateInterest() {
    return balance * interestRate;
}
```

The method `calculateInterest()`uses both the *instance* variable `balance` and *class* variable `interestRate`, which means that it calculates the interest based on its own attribute `balance` and the common attribute `interestRate` that is shared by all `Account` objects. It can be visualized in the following diagram (Figure 3.24):

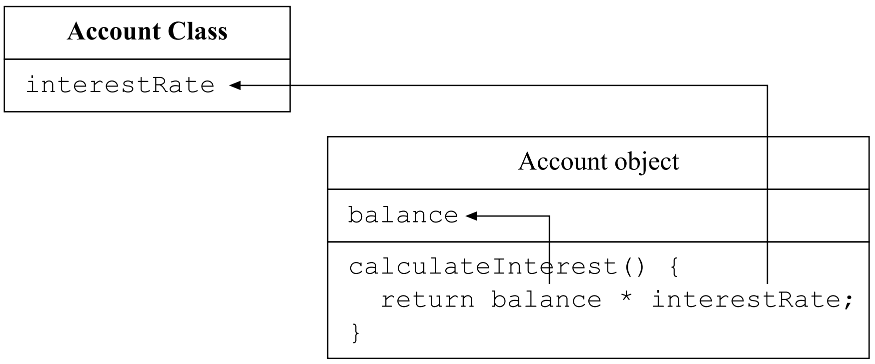


Figure 3.24 Class variable and instance variable

The arrows in the above diagram show what each variable name is referring to. If there are two `Account` objects, the scenario is shown in the following diagram:

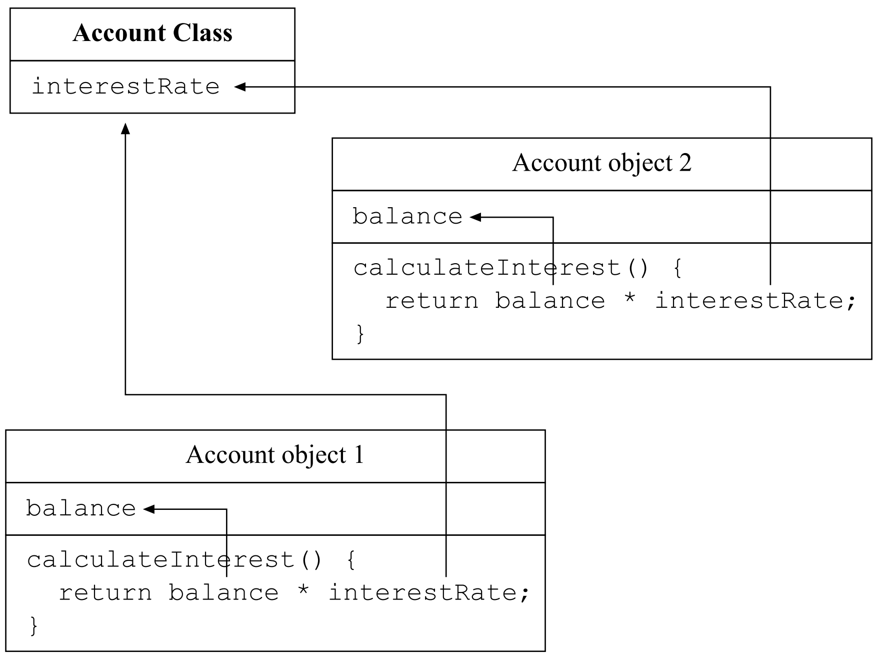


Figure 3.25 Class variable and instance variables of two different objects

Using class members

If a class has class members including variables and methods, they can be used as if they are usual members. For example, in the method `calculateInterest()` that was discussed, you can write an expression including an instance variable and a class variable without any special indication. The computer will determine whether it is an instance variable or a class variable.

A core difference between instance members and class members is that class members exist without the necessity of creating an object of the class. However, instance members exist when you create the object. For example, a bank has an interest rate regardless of whether there are many accounts opened or no one has yet opened an account. Therefore, it is possible to access a class member before creating an object based on the class. The following reading helps you understand more about class variables.

Reading

King, Section 7.5, pp. 274–77

From the reading, you know most class variables should be declared `private` similar to instance variables, unless it is necessary to make them accessible by objects of other classes. Furthermore, the reading says that the Java keyword `final` can be used to mark a variable to be constant or immutable, so that the content of the variable cannot be changed (the use of the keyword is discussed in a later unit). The aforementioned class `Account` can be modified to be:

```
class Account {
  public static final double INTEREST_RATE = 5.0;
  ......
}
```

For such kinds of constant variable, the identifiers are all in uppercase, such as `LIMIT`. If the constant variable contains more than one word, the words are separated by an underscore, such as `INTEREST_RATE and DISK_QUOTA`.

The reading also mentions a class `Math` and two class variables were defined — `PI` and `E`. As these two class variables are marked public, any objects in the JVM can access them. You can use `Math.PI` and `Math.E` to access them respectively. By the dot-notation, it seems that the Math class is an object; both `PI` and `E` are its attributes.

Actually, a class definition in the JVM can be considered an object. This object is created automatically when the JVM loads the class definition from the corresponding class file. A significant difference between a usual object and class object is that there must be only one class object for a class definition, whereas a single class definition can create more than one object.

Class variables can be treated as the attribute of the class object. Like usual objects, a class object can also have its own set of methods, known as class methods. Like class variables, you can also define a class method using the `static` keyword. For example, you can have a method `calculateCircumference()` to calculate the circumference of a circle:

```
// The class definition of a simple calculator
class Calculator {
  // To calculate the circle circumference
  public static double calculateCircumference(double r) {
    return 2.0 * 3.1415 * r;
  }
}
```

Similar to class variables, it is possible to use a class method without creating an object of the class. Therefore, you can have the following expression

```
double c = Calculator.calculateCircumference(2.0);
```

to calculate the circumference of a circle with a radius of two meters. To execute a class method, you send a message to the class definition instead, such as in the above statement.

A special class method named `main()` is crucial to a Java application. The definition of the method is:

```
public static void main(String args[]) {
. . . . . .
}
```

You can see that the requirements of the method `main()` are:

1. It must be marked `public`, which means that this method can be accessed by the JVM.

2. It must be marked `static`. As mentioned, class methods can be accessed even if no object of the class has been created. When the JVM loads the class definition, it will never create an object of the class immediately to start the method `main()`. Therefore, the method `main()` has to be a class method, so that the JVM can execute it without creating an object of the class.

3. As the method returns nothing, you have to specify the return type as `void`.

4. The method parameter list must be an array of `String` objects (`String[]`).*Unit 5* discusses the use of arrays.

For example, the following is the first program you encountered in *Unit 1*:

```
public class HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello World");
  }
}
```

The significance of this class method `main()` is that it is the method to be executed when the JVM loads the class `HelloWorld` and runs it. Therefore, the first statement in the method `main()` is the first statement to be executed when your program starts. When the execution of the class method `main()` is terminated, the JVM usually terminates.

The difference in the lifetime of instance methods and class methods can help us understand an issue of accessibility to instance and class members. Let's consider a class definition that declares both instance and class members (variables and methods). When the JVM loads this class definition, all class variables and methods are available. If your program executes the class's class methods at this moment, the members that can be accessed are only class variables and methods (and the local variables of the method).

However, if your program creates an object of the class, all instance variables and methods are available and are associated with the object. Then, if you execute the instance method, both instance members (of the object) and class members (of the class) are available.

In a few words, instance methods can access both instance and class members, whereas class methods can only access class members. The following table shows you the differences between instance and class members (variables and methods).

	Instance members	**Class members**
Declaration	Without keyword `static`	With keyword `static`
Associated with	An object of the class (for instance variables, each object has its own copy).	The class itself (all objects of the same class share the same class variable).
Available since	The creation of an object.	The class definition is loaded into the JVM.
Access via	A reference variable of the class type.	Usually the class name.
Accessibility	Accessible by instance methods only.	Accessible by both instance and class methods.

Please notice that class members can also be accessed via any object of the class. That is,

reference-variable.class-member

For example, the class variable, `INTEREST_RATE`, of the class `Bank` can be accessed by

```
double rate = Bank.INTEREST_RATE;
```

or

```
Bank b = new Bank();
double rate = b.INTEREST_RATE;
```

The former convention is preferable. Not only is it shorter, it helps us recognize that the variable `INTEREST_RATE` is associated with the class `Bank`, instead of with a particular object of the class `Bank` as in the latter one.

Because you can execute a class method without creating an object of the class beforehand, class methods are usually used as utility methods, such as the `calculateCircumference()` method of the class `Calculator`. Another excellent example of utility methods is the methods of the class `Math`.

The class `Math` defines a set of common mathematical methods that you need in scientific calculations. The following table shows some of the pre-defined methods.

Category	**Methods**
Trigonometric	`public static double acos(double a)` returns the arc cosine of an angle `public static double asin(double a)` returns the arc sine of an angle `public static double atan(double a)` returns the arc tangent of an angle `public static double cos(double a)` returns the cosine of an angle `public static double sin(double a)` returns the sine of an angle `public static double tan(double a)` returns the tangent of an angle
Exponential/ Logarithmic	`public static double exp(double a)` returns the result of e^a where *e* is the base of natural logarithms `public static double log(double a)` returns the natural logarithm of a number. `public static double pow(double a, double b)` returns the result of a^b
Random number	`public static double random()` returns a random number greater than or equal to 0 and less than 1
Miscellaneous	`public static double abs(double a)` returns the absolute value of a number `public static double min(double a, double b)` returns the smaller number of the two numbers `public static double max(double a, double b)` returns the larger number of the two numbers

All methods defined in the class `Math` are class methods. Therefore, you can use any method by specifying the class name `Math`. For example, you can use the following statement to get a random number from 0 to 9 inclusive:

```
int num = (int) (Math.random() * 10.0);
```

Summary

This unit starts with different types of object that we derived from *Unit 2*. Then we discussed the way to represent these attributes and behaviours as instance variables and instance methods respectively in class definitions. With the class definitions, we can create objects based on the class definitions to model real-world entities and situations so that we can solve the problems by manipulating the object attributes and behaviours.

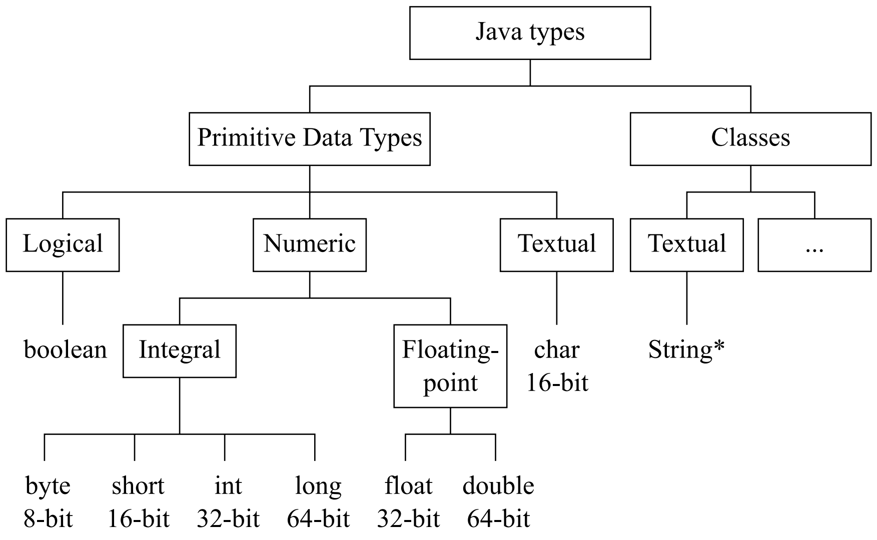
You define attributes in a class using the following syntax:

variable-type variable-name`;`

To define a method, the syntax is:

```
return-type method-name([parameter-list]) {
   [statements]
}
```

We then further discussed the types of data that your Java programs can manipulate. There are two types of category for the Java programming language — primitive and non-primitive (or classes). The following chart can help you learn the different types.



Based on your understanding of the object attributes, you can select suitable types and assign them to the attributes. In the above chart, the `String` type is added to the diagram to highlight the variable types that can store a single character and a sequence of characters are `char` and `String` types respectively. Please be aware that `String` is not a primitive type — it is a non-primitive type.

Occasionally, a few attributes can be grouped to form structured data, such as the `Date` non-primitive type discussed earlier.

While handling data, you should take extra care to make sure that the variable on the left-hand side of the equals sign is assignment compatible with the value of the expression on the right-hand side of the equals sign. Otherwise, you need to use casting to explicitly convert a numeric type to another type.

Finally, we discussed operations on numeric data including addition, subtraction, multiplication, division and their order. Multiplication and division is always performed before addition and subtraction. You can use parentheses to override the operation orders if necessary.

Suggested answers to self-test questions

Self-test 3.1

The following is a list of valid identifiers: mt201, java_is_easy, balance4Account, AccountBook and Class.

12aDozen and interface are invalid identifiers. For 12aDozen, an identifier cannot start with a digit, and interface is a keyword in Java and cannot be used as an identifier. Note, however, that Interface (like Class above) would be acceptable as an identifier *but you are strongly advised never to use keywords in any format as identifiers.*

Self-test 3.2

1 The `Date` class can be defined as:

```
class Date {
  int day;
  int month;
  int year;
}
```

2 The class for storing student examination results can be defined as:

```
class StudentResult {
  double chineseScore;
  double englishScore;
  double mathScore;
}
```

3 The class or Computer can be defined as:

```
class Computer {
  double cpuSpeed;
  double hardDiskSize;
  double memorySize;
  double cdROMSpeed;
  double price;
}
```

Self-test 3.3

The `Account` class can be defined as:

```
// The class definition of an account
class Account {
  double balance;        // The account balance

  // To deposit the specified amount to the account
  void deposit(double amount) {
    balance = balance + amount;
  }
  // To withdraw the specified amount from the account
  void withdraw(double amount) {
    balance = balance – amount;
  }
}
```

Self-test 3.4

1 If you run the program, the `new` operation is executed twice. Therefore, two objects are created.

2 The statement ‘`counter2 = counter1;`’, with the use of the little box model, means that the message in the box labeled `counter1` is copied and placed in the box labeled `counter2`. The result is that the original message in the box labeled `counter2` is lost. Then, it is not possible to find the ticket counter that was originally described by the message in the box labeled. The original ticket counter referred to by `counter2` is lost and it can never be located again.

3 After the statement ‘`counter2 = counter1;`’, the messages in both little boxes labeled `counter1` and `counter2` are the same. Therefore, following the message in the box labeled `counter2` gets the ticket counter and changes its attribute to 3. Afterward, you follow the message in the box labeled `counter1` to get the ticket counter, and you will find that the value referred by `counter1` is 3.

Self-test 3.5

```
TicketCounter counter1, counter2;
```

Two little boxes labeled `counter1` and `counter2` respectively are prepared:

```
counter1 = new TicketCounter();
```

A ticket counter is available, and you store the way to get it in the little box labeled `counter1`:

```
counter2 = counter1;
```

You copy the message in the little box labeled `counter1` and store it in the little box labeled `counter2`:

```
counter1.setReading(10);
```

You get the ticket counter by following the way stored in the little box labeled `counter1` and set its attribute `reading` to 10.

```
counter1.increase();
```

Similarly, you get the ticket counter by following the way stored in the little box labeled `counter1` and send a message `increase` to it, such as by pressing a button on it. Then, the ticket counter increases the value of its own `reading` to 11.

```
counter2.increaseByAmount(2);
```

You get the ticket counter by following the way stored in the little box labeled `counter2` and send a message `increaseByAmount` and an extra data ‘two’ to it. Therefore, the ticket counter increases its own `reading` by 2 to 13.

At the end, the value of `reading` is 13, because by following either of the ways in the little boxes labelled `counter1` and `counter2`, you end up at the same ticket counter.

Self-test 3.6

The results of the expressions are:

1 = 1 – 2 + 3
= -1 + 3
= 2

2 = 3 * 4 / 2
= 12 / 2
= 6

3 = (1 + 3) * 4 / 3
= 4 * 4 / 3
= 16 / 3
= 5 (integer division)

4 = 4 * 4 / 2 / 2
= 16 / 2 / 2
= 8 / 2
= 4

Self-test 3.7

1 The assignment needs casting, because `char` can be considered a 16-bit integral type, whereas `int` is a 32-bit integral type.

2 It needs casting, because `float` and `double` are 32-bit and 64-bit floating-point types respectively, and `float` is not large enough to store a 64-bit floating-point value.

3 Casting is not necessary, because `int` is large enough to store a `byte` value.

4 Even though both `char` and `short` are 16-bit integral types, the assignment needs casting, because their ranges are different.