CHAPTER 7

Defining classes

LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- **How to define your own classes that implement abstractions.**
- **How to declare, initialize and use field variables.**
- **How to declare and call methods.**
- **How to pass information to methods and how methods return values.**
- **How to use the `this` reference to refer to the current object.**
- **How static members of a class are declared and used.**
- **How information is passed to the program via program arguments.**
- **How to initialize the state of newly-created objects using constructors.**
- **How enumerated types can be used to define finite sets of symbolic constants.**

INTRODUCTION

Chapter 4 introduced the object model that is the foundation for object-oriented programming, in which programs are composed of objects that collaborate to provide the required functionality. An object belongs to a class that defines the common properties and behaviour of a particular type of objects. When writing programs, defining and understanding problem-specific classes is essential. In this chapter we focus on how to define and use our own classes, called user-defined classes, as opposed to the predefined classes found in the Java standard library. We also introduce a special kind of class that can be used to define finite sets of symbolic constants, called enumerated types.

7.1 Class members

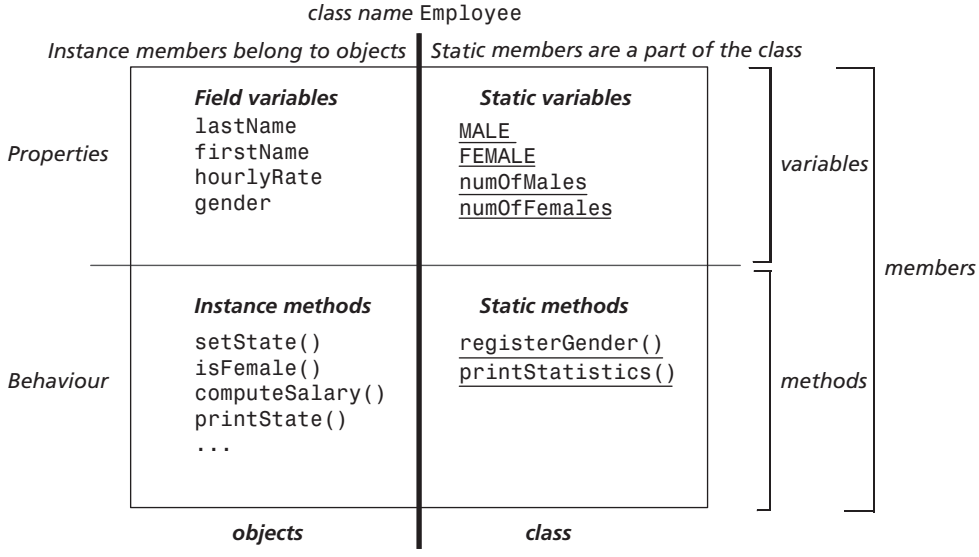
A *class declaration* defines the properties and the behaviour of the objects of the class. We will discuss the purpose of the different declarations that can be specified in a class declaration. We will also look at how these declarations are used inside the class and by other classes. These declarations are called *member declarations*.

Figure 7.1 gives an overview of the different *members* that can be declared in a class. *Field variables* represent properties and *instance methods* define the behaviour of the *objects* of the class. The term *instance* means an object of a class. Field variables and instance methods are collectively called *instance members*, and belong to the objects of the class.

In Java, a class can also define the properties and behaviour that belong to the class (see Figure 7.1). *Static variables* represent properties, and *static methods* define the behaviour of the class. Static variables and static methods are collectively called *static members*, and belong to the *class*, *not* to the objects of the class. Static members are discussed in Section 7.4.

Member declarations in a class can be declared in any order. It's common practice to group instance and static members separately, and further organize them according to fields and methods, as shown in Figure 7.1.

FIGURE 7.1 Overview of members in a class declaration



In addition to the members mentioned above, a class can also declare *constructors*. Constructors resemble methods, but the primary use of a constructor is to set the state of an object when the object is created. Section 7.5 discusses constructors.

We will use a class for employees in a company as a running example, and fill in the details of the class declaration in subsequent sections. You may find it useful to refer to

Figure 7.1 as we discuss the various members, to help identify to which group a member belongs.

7.2 Defining object properties using field variables

Field declarations

As we have noted, field variables in a class declaration define the properties of objects that can be created from the class. Each object gets its own copy of the field variables. A *field declaration* specifies both the *field type* and the *field name*, just like in the declaration of local variables. The code below defines four field variables for the class `EmployeeV1`:

```
class EmployeeV1 {              // Assume that no constructors are declared.
  // Field variables
  String  firstName;
  String  lastName;
  double  hourlyRate;
  boolean gender;               // false means male, true means female
  // ...
}
```

The field declarations above show that a field can be either a variable, such as the field name `hourlyRate`, that can store a value of a primitive type, for example the primitive type `double`, or it can be a reference variable, such as the field name `firstName`, that can store a reference value of an object, for example that of the `String` class. Objects that are created from the class will have a field for each of the field declarations in the class declaration.

Initializing fields

We create objects from a class using the `new` operator, which requires a constructor call:

```
EmployeeV1 anEmployee = new EmployeeV1();
```

The `new` operator creates a new object of the `EmployeeV1` class in memory. This object will have room for all the fields declared in the class `EmployeeV1`. In the current version of the class, all objects created this way will have their fields initialized to *default values*, as shown in Figure 7.2a.

The *state* of an object comprises all values in the field variables of the object at any given time. The state can change over time, as the field values change. In the case of the `EmployeeV1` class, the *initial state* of an object is thus the default values of the field variables (Figure 7.2a).

FIGURE 7.2 Field initialization (no constructors defined)

(a) Field initialization with default values

(b) Field initialization with initial values

It is possible to assign an initial value to the variable in a variable declaration. The same can be done for field variables. The class `EmployeeV2` specifies an initial value for all its field variables:

```
class EmployeeV2 {              // Assume that no constructors are declared.
  // Field variables with initial values
  String  firstName  = "Joe";
  String  lastName   = "Jones";
  double  hourlyRate = 15.50;
  boolean gender     = false;  // false means male, true means female
  // ...
}
```

We can create an object of the class `EmployeeV2` using the `new` operator:

```
EmployeeV2 employeeA = new EmployeeV2();
```

Execution of the expression `new EmployeeV2()` will always return an object with the initial state shown in Figure 7.2b, assuming that the class does not define any constructors that initialize the fields. Section 7.5 explains how the initial state can be customised by using constructors.

BEST PRACTICE

Always initialize the fields of a newly-created object in a constructor, so that the object has a valid initial state.

7

7.3 Defining behaviour using instance methods

Method declaration and formal parameters

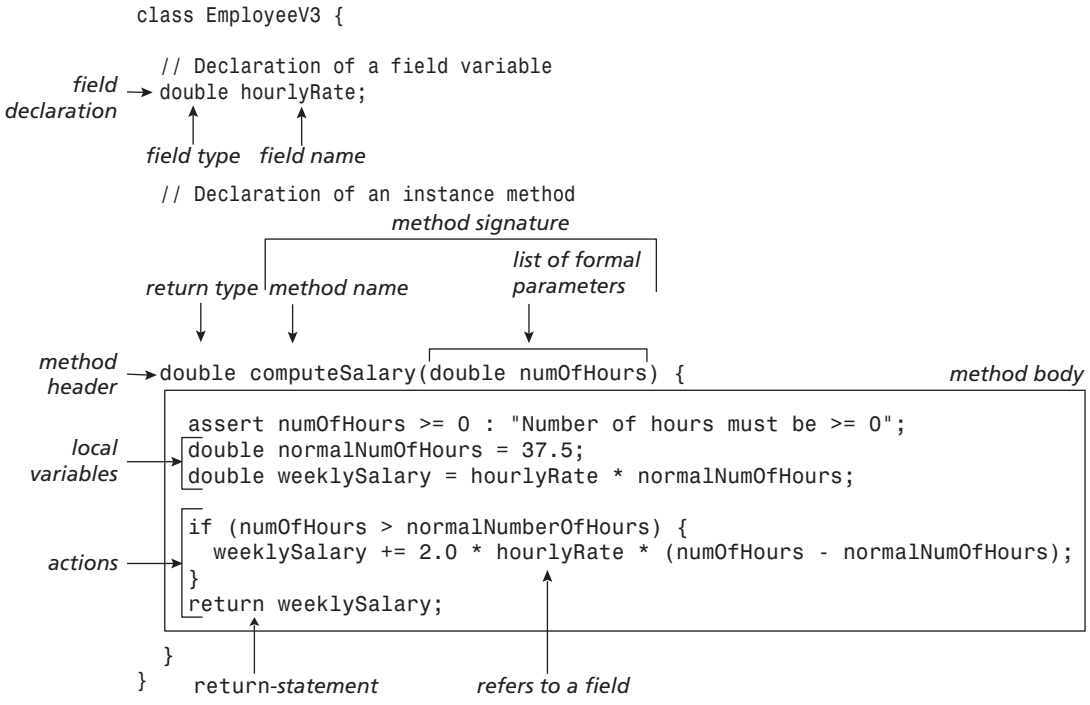
In Chapter 3 we saw that a method declaration comprises a method header and a method body. Figure 7.3 shows the declaration of the method `computeSalary()` in the class `EmployeeV3`. The method header declares the *return type*, the *method name* and the *parameter list*.

The return type of the method `computeSalary()` is the primitive type `double`, which means that the method returns a floating-point number when the method is called. If a method is not supposed to return a value, the keyword `void` should be specified instead of the return type. For example, see the method `setState()` in class `EmployeeV3` shown in Program 7.1. A non-`void` method must specify a return type.

The name of the method reflects what it does. The parameter list of the method indicates what information the method needs to do its job. The parameter list declares *formal parameters* for the method, in which each parameter is specified as a variable declaration with the parameter name and the parameter type. The method `computeSalary()` has a formal parameter `numOfHours` with the primitive type `double`. The type of a formal parameter can also be a reference type, for example a class or an array. The parameter list is always enclosed in parentheses, `()`, even if the method has no parameters. The method and the parameter list constitute the *signature* of the method. The signature determines which method declaration is chosen for execution by a method call. The method `computeSalary()` has the following signature:

```
computeSalary(double)
```

FIGURE 7.3 Method declaration



The method body comprises variable declarations and actions. Variable declarations in the method body define the local variables needed to hold values during the execution of the method. Such variables are accessible only in the method body, and are not accessible outside the method in which they are declared. Formal parameters are also local variables. Several methods can have the same names for their local variables, but these are only

accessible within the method in which they are declared. The method `computeSalary()` has the following local variables: `normalNumOfHours`, `numOfHours`, `weeklySalary`.

The method body implements the actions. For example, the `if` statement in the body of the method `computeSalary()` is used to determine whether the employee that the object represents should receive any compensation for overtime. There are several statements, such as assignments, control flow statements and method calls, that can be used when implementing the behaviour of objects.

Local variables and statements can be defined in any order, but the rule is that a local variable must be declared before it can be used in the method body. It is a good idea to declare a local variable at the same time as when first assigning a value to it. This aids in making the purpose of the program clear.

PROGRAM 7.1 Declaration of instance methods

```
class EmployeeV3 {              // Assume that no constructors are declared.
  // Field variables
  String firstName;
  String lastName;
  double hourlyRate;
  boolean gender;               // false means male, true means female

  // Instance methods

  // Assign values to the field variables of an employee.
  void setState(String fName, String lName,
                double hRate, boolean genderValue) {
    firstName = fName;
    lastName = lName;
    hourlyRate = hRate;
    gender = genderValue;
  }

  // Determines whether an employee is female.
  boolean isFemale() { return gender; }

  // Computes the salary of an employee, based on the number of hours
  // worked during the week.
  double computeSalary(double numOfHours) {
    assert numOfHours >= 0 : "Number of hours must be >= 0";
    double normalNumOfHours = 37.5;
    double weeklySalary = hourlyRate * normalNumOfHours;
    if (numOfHours > normalNumOfHours) {
      weeklySalary += 2.0 * hourlyRate * (numOfHours - normalNumOfHours);
    }
    return weeklySalary;
  }

  // Prints the values in the field variables of an employee.
```

```
  void printState() {
    System.out.print("First name: " + firstName);
    System.out.print("\tLast name: " + lastName);
    System.out.printf("\tHourly rate: %.2f", hourlyRate);
    if (isFemale()) {
      System.out.println("\tGender: Female");
    } else {
      System.out.println("\tGender: Male");
    }
  }
}
```

Method calls and actual parameter expressions

A method call is used to execute a method body. Figure 7.4 shows an example. A call specifies the object whose method is called (given by the reference `manager` in this case), the name of the method and any information the method needs to execute its actions. These are referred to as *actual parameters*, or *arguments*. An actual parameter is an *expression*, in contrast to a formal parameter specified in the method declaration, which is always a *variable*. The *signature of a method call* consists of the method name and the type of the actual parameter expressions. In Figure 7.4, for example, the method call has the following signature:

```
setState(String, String, double, boolean)        // (1) Call signature
```

The compiler checks that a method exists that corresponds to the method call. It requires the signature of the method call to be compatible with the signature of the method declaration. This compatibility implies that the value of the first actual parameter can be assigned to the first formal parameter, the value of the second actual parameter can be assigned to the second formal parameter and so on. In Figure 7.4 the declaration of the method `setState()` has the following signature:

```
setState(String, String, double, boolean)       // (2) Method signature
```

We can therefore see that the signature of the call to the method `setState()` at (1) corresponds to the signature of the method declaration at (2). For all formal parameters, the value of the actual parameter is assigned to the corresponding formal parameter variable, as shown in Figure 7.4.

Finally, here are some examples of method calls that result in compile-time errors. The following method calls to the method `setState()`:

```
manager.setState(name, hourlyRate*2.0, "Jones", false); // (3) Method call
manager.setState(name, "Jones", hourlyRate*2.0);        // (4) Method call
```

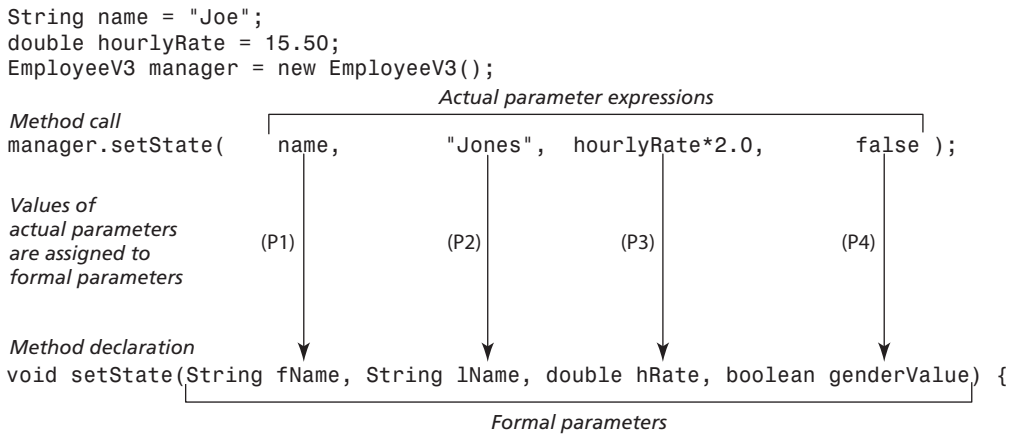
have the following signatures respectively:

```
setState(String, double, String, boolean)  // (5) Call signature
setState(String, String, double)           // (6) Call signature
```

We can see that the call signatures at (5) and (6) do not correspond to the method signature in (2) above. The call signature at (5) results in a compile-time error, because of the

second parameter in the call, as a value of type `double` cannot be assigned to a `String` reference in the method signature. In the call signature at (6), the number of actual parameters is not correct – it should be four.

FIGURE 7.4 Parameter passing



After parameter passing:
(P1) `fName` *and* `name` *are aliases of the* `String` *object with the value* `"Joe"`.
(P2) `lName` *refers to the* `String` *object with the value* `"Jones"`.
(P3) `hRate` *has the value* `31.0`, *i.e.* `hourlyRate*2.0 = 15.50*2.0`.
(P4) `genderValue` *has the value* `false`.

Parameter passing: call-by-value

An actual parameter expression is evaluated and its value is assigned to the corresponding formal parameter variable. All actual expressions are evaluated before the call to the method is executed. In the call to the `setState()` method in Figure 7.4, all actual parameters are simple expressions that evaluate to the following values: a reference value of a `String` object with the value `"Joe"`, a reference value of a `String` object with the value `"Jones"`, the floating-point value `31.0` (`hourlyRate*2.0 = 15.5*2.0`), and the Boolean value `false`.

Before the method is executed, values of the actual parameter expressions are assigned to the corresponding formal parameter variables. Parameter passing in Figure 7.4 is equivalent to the following assignments:

```
String fName = name;          // (P1) reference value of "Joe"
String lName = "Jones";       // (P2) reference value of "Jones"
double hRate = 31.0;          // (P3) primitive value 31.0
boolean genderValue = false;  // (P4) primitive value false
```

At (P1) and (P2) in Figure 7.4, the actual parameter values are *reference values*, so these reference values are passed before execution of the method `setState()` starts. The assign-

ment at (P1) implies that references `fName` and `name` are aliases to the same `String` object with the value `"Joe"` at the start of execution of the method `setState()`. Analogously, the reference `lName` at (P2) refers to the `String` object with the value `"Jones"` at the start of the execution in the method `setState()`. Note that no objects are copied, only reference values. (P3) and (P4) show the passing of primitive values, `double` and `boolean` respectively. This way of passing parameters in which only values are passed is called *call-by-value*.

Program 7.2 illustrates the execution of the method call in Figure 7.4. The program prints the state of the `EmployeeV3` object referred to by the `manager` reference before and after the call to the `setState()` method. The program output confirms that the state of the `EmployeeV3` object created in (1) was updated with the values of the actual parameter expressions in the method call at (2).

PROGRAM 7.2 Parameter passing: call-by-value

```
// Illustrating parameter passing
public class Client3A {
  public static void main(String[] args) {

    String name = "Joe";
    double hourlyRate = 15.50;
    EmployeeV3 manager = new EmployeeV3();                    // (1)
    System.out.println("Manager state before call to setState() method");
    manager.printState();
    manager.setState(name, "Jones", hourlyRate*2.0, false);  // (2)
    System.out.println("Manager state after call to setState() method");
    manager.printState();
    System.out.println();

    System.out.printf("Manager hourly rate before adjusting: %.2f%n",
                      manager.hourlyRate);                    // (3)
    adjustHourlyRate(manager.hourlyRate);     // (4) LOGICAL ERROR!
    System.out.printf("Manager hourly rate after adjusting: %.2f%n",
                      manager.hourlyRate);
    System.out.println();

    EmployeeV3 director = new EmployeeV3();                   // (5)
    System.out.println("Director state before call to copyState() method");
    director.printState();
    copyState(manager, director);                             // (6)
    assert (manager.lastName.equals(director.lastName) &&
            manager.firstName.equals(director.firstName) &&
            manager.hourlyRate == director.hourlyRate &&
            manager.gender == director.gender) :
            "Manager and director have different states after copying.";
    System.out.println("Director state after call to copyState() method:");
    director.printState();
    System.out.println("Manager state after call to copyState() method:");
```


```
    manager.printState();
    System.out.println();
  }

  // Method that tries to adjust the hourly rate.
  static void adjustHourlyRate(double hourlyRate) {          // (7)
    hourlyRate = 1.5 * hourlyRate;                           // (8)
    System.out.printf("Adjusted hourlyRate: %.2f%n", hourlyRate);
  }

  // Method that copies the state of one employee over to another employee.
  static void copyState(EmployeeV3 fromEmployee,
                        EmployeeV3 toEmployee) {             // (9)

    toEmployee.setState(fromEmployee.firstName,              // (10)
                        fromEmployee.lastName,
                        fromEmployee.hourlyRate,
                        fromEmployee.gender);

    toEmployee = fromEmployee = null;                        // (11)
  }
}
```

Program output:

```
Manager state before call to setState() method
First name: null  Last name: null Hourly rate: 0.00 Gender: Male
Manager state after call to setState() method
First name: Joe Last name: Jones  Hourly rate: 31.00  Gender: Male

Manager hourly rate before adjusting: 31.00
Adjusted hourlyRate: 46.50
Manager hourly rate after adjusting: 31.00

Director state before call to copyState() method
First name: null  Last name: null Hourly rate: 0.00 Gender: Male
Director state after call to copyState() method:
First name: Joe Last name: Jones  Hourly rate: 31.00  Gender: Male
Manager state after call to copyState() method:
First name: Joe Last name: Jones  Hourly rate: 31.00  Gender: Male
```

7

Consequences of call-by-value

Inside the method body a formal parameter is used like any other local variable in the method, so that changing its value in the method has no effect on the value of the corresponding actual parameter in the method call.

The class `Client3A` defines a static method `adjustHourlyRate()` at (7) in Program 7.2. This method changes the value of the formal parameter `hourlyRate`:

```
  hourlyRate = 1.5 * hourlyRate;                             // (8)
```

The method `adjustHourlyRate()` is called at (4) in the hope of adjusting the hourly rate of an employee. Output from the program shows that changing the value of the formal parameter `hourlyRate` in the method has no effect on the value of the variable `manager.hourlyRate`, which is the corresponding actual parameter. A simple solution for getting the adjusted value from the method `adjustHourlyRate()` is to modify the method so that it returns the adjusted value:

```
static double adjustHourlyRate(double hourlyRate) {              // (7)
  hourlyRate = 1.5 * hourlyRate;                                 // (8)
  System.out.printf("Adjusted hourly rate: %.2f%n", hourlyRate);
  return hourlyRate;
}
```

The value returned by the method call can then be assigned explicitly:

```
manager.hourlyRate = adjustHourlyRate(manager.hourlyRate);
```

Reference values as actual parameter values

The state of an object whose reference value is passed to a formal parameter variable can be changed in the method, and the changes will be apparent after return from the method call. Such a state change is illustrated by the method `copyState()` in Program 7.2.

The class `Client3A` defines the static method `copyState()` at (9), which copies the state of one employee to another employee. The method is called at (6):

```
copyState(manager, director);                                    // (6)
```

After the call we see from the output that the state of the `EmployeeV3` object referred to by the reference `director` has the same state as the `EmployeeV3` object referred to by the reference `manager`. Only the reference values of the two objects are passed in the method call. The state of the employee object referred to by the reference `director` is changed via the formal parameter reference `toEmployee`. These changes are apparent after return from the method call, as confirmed by the program output. At (11) the values of both the formal variables are set to `null`, but this has no effect on the variables that make up the actual parameter expressions. These variables, `manager` and `director`, still refer to their respective objects after return from the method.

Arrays as actual parameter values

Passing arrays as parameter values does not differ from what we have seen earlier for other objects that occur in actual parameter expressions. If the actual parameter evaluates to a reference value of an array, for example, then this reference value is passed. Program 7.3 illustrates the use of arrays as actual parameters.

Four arrays are created with information about three employees at (1) in Program 7.3. The same index in all the four arrays gives information about the same employee. This information is used to create an array of three employees in (2), (3) and (4). In addition, an array contains the number of hours each employee has worked in a week, at (5). Finally, the salaries of all the employees in the array `employeeArray` are computed by a call to the static method `computeSalaries()`.

Because we use the `[]` notation to declare an array, we use the same notation to declare an array reference as a formal parameter. An example of such a declaration is shown at (7) in Program 7.3:

```
static void computeSalaries(EmployeeV3[] employeeArray,
                            double[] hoursArray) {...}
```

The references `employeeArray` and `hoursArray` refer to arrays of type `EmployeeV3[]` and `double[]`, respectively. Without the `[]` notation, these references would not be array references. (6) shows a call to this method:

```
computeSalaries(employeeArray, hoursArray);
```

Again parameter passing is equivalent to the following assignments in which the reference values of the arrays are passed:

```
EmployeeV3[] employees = employeeArray;     // Reference value of array
double[]     hours     = hoursArray;        // Reference value of array
```

We don't use the `[]` notation for array variables when these are passed as actual parameters in a method call. Instead, the reference value of the array is passed just like the reference value of any other object, as shown in the assignments above. Note that the signature of the method is:

```
computeSalaries(EmployeeV3[], double[])      // Method signature
```

and the signature of the method call corresponds to the method signature. Note also that there is no requirement on the length of the array in the method call, which is implicitly given by the `length` field of the array. However, the call must have the right *array type*.

If an *array element* occurs in an actual parameter expression, the value of the element is used in the evaluation of the expression and the expression value is passed, as one would expect. If the actual parameter is an array element of a primitive type, for example `hourlyRateArray[2]`, whose type is `double`, the primitive value is passed. If the actual parameter is an array element of a reference type, for example `lastNameArray[1]`, whose type is `String`, the reference value of the element object is passed. (4) in Program 7.3 shows a call to the method `setState()`, in which the actual parameter expressions are array elements. Note that the types of the actual parameters at (4) are in agreement with the type list (`String, String, double, boolean`) in the signature of the `setState()` method.

PROGRAM 7.3 Arrays as actual parameters

```
// Passing arrays
public class Client3B {

  public static void main(String[] args) {
    // (1) Associated arrays with information about employees:
    String[] firstNameArray = { "Tom", "Dick", "Linda" };
    String[] lastNameArray = { "Tanner", "Dickens", "Larsen" };
    double[] hourlyRateArray = { 30.00, 25.50, 15.00 };
    boolean[] genderArray = { false, false, true };
```

```
    // (2) Array with employees:
    EmployeeV3[] employeeArray = new EmployeeV3[3];

    // (3) Create all employees:
    for (int i = 0; i < employeeArray.length; i++) {
      employeeArray[i] = new EmployeeV3();
      employeeArray[i].setState(firstNameArray[i], lastNameArray[i],
                                hourlyRateArray[i], genderArray[i]);  // (4)
    }

    // (5) Array with hours worked by each employee:
    double[] hoursArray = { 50.5, 32.8, 66.0 };

    // (6) Compute the salary for all employees:
    computeSalaries(employeeArray, hoursArray);
  }

  // (7) Compute the salary for all employees:
  static void computeSalaries(EmployeeV3[] employees,
                              double[] hours) {
    for (int i = 0; i < employees.length; i++) {
      System.out.printf("Salary for %s %s: %.2f%n",
                        employees[i].firstName,
                        employees[i].lastName,
                        employees[i].computeSalary(hours[i]));
    }
  }
}
```

Program output:

```
Salary for Tom Tanner: 1905.00
Salary for Dick Dickens: 956.25
Salary for Linda Larsen: 1417.50
```

The current object: `this`

When we call an instance method of an object, we say that the method is *invoked* on the object. The object whose method is invoked becomes the *current object*. Inside the method, the current object can be referred to by the keyword `this`, i.e. the keyword `this` is a reference. We can think of the `this` reference as an implicit actual parameter that is passed to an instance method each time it is invoked. For example, in the method call:

```
manager.setState(name, "Jones", hourlyRate*2.0, false);  // (2)
```

the reference value held in the reference `manager` will be available in the `this` reference inside the method `setState()` when the method is executed. So the method `setState()` in Program 7.1 could be declared as follows using the `this` reference:

```
void setState(String fName, String lName,
              double hRate, boolean genderValue) {
  this.firstName  = fName;
  this.lastName   = lName;
  this.hourlyRate = hRate;
  this.gender     = genderValue;
}
```

The reference `this` can be used exactly like any other reference in the method, except that its value cannot be changed, i.e. it is a `final` reference. For example, we can refer to the field `firstName` in the current object by using the expression `this.firstName`, as shown above in the declaration of the method.

If a local variable has the same name as a field variable, the local variable will *shadow* the field variable, as shown in this declaration of the same method:

```
void setState(String firstName, String lastName,
              double hRate, boolean genderValue) {
  firstName  = firstName;  // (1)
  lastName   = lastName;   // (2)
  hourlyRate = hRate;
  gender     = genderValue;
}
```

The first two formal parameters have the same names as the field variables `firstName` and `lastName`. Formal parameters are local variables, and inside the method these two names refer to the formal parameters. At (1) and (2) above, values of the formal parameters are assigned back to the formal parameters. We can use the `this` reference to distinguish the field variable from the local variable when they have the same name:

```
void setState(String firstName, String lastName,
              double hRate, boolean genderValue) {
  this.firstName  = firstName;  // (1')
  this.lastName   = lastName;   // (2')
  this.hourlyRate = hRate;
  this.gender     = genderValue;
}
```


Method execution and the `return` statement

When we invoke a method on an object, the method can in turn invoke other methods, either on the current object or on other objects. Figure 7.5 on page 176 illustrates invocation of the method `setState()` in which we have introduced two local variables (`i` and `s`) and two calls to the method `printState()`:

```
void setState(String firstName, String lastName,
              double hRate, boolean genderValue) {
  // Some superfluous local variables
  int i;
  String s;

  this.printState();             // Print state before
  this.firstName  = firstName;
  this.lastName   = lastName;
  this.hourlyRate = hRate;
  this.gender     = genderValue;
  this.printState();             // Print state after
}
```

The call to the method `setState()` is not complete when the executions of the two calls to the method `printState()` has completed. The current object is an object of class `EmployeeV3`, and its state will be printed twice. Note that execution continues in the method `setState()` after return from the calls to the method `printState()`. The execution of the calls to the method `printState()` is embedded in the call to the method `setState()`.

A local variable is not initialized to a default value during method execution. If we try to use a local variable before it has been assigned a value, the compiler will report an error.

BEST PRACTICE

Always initialize local variables when they are declared.

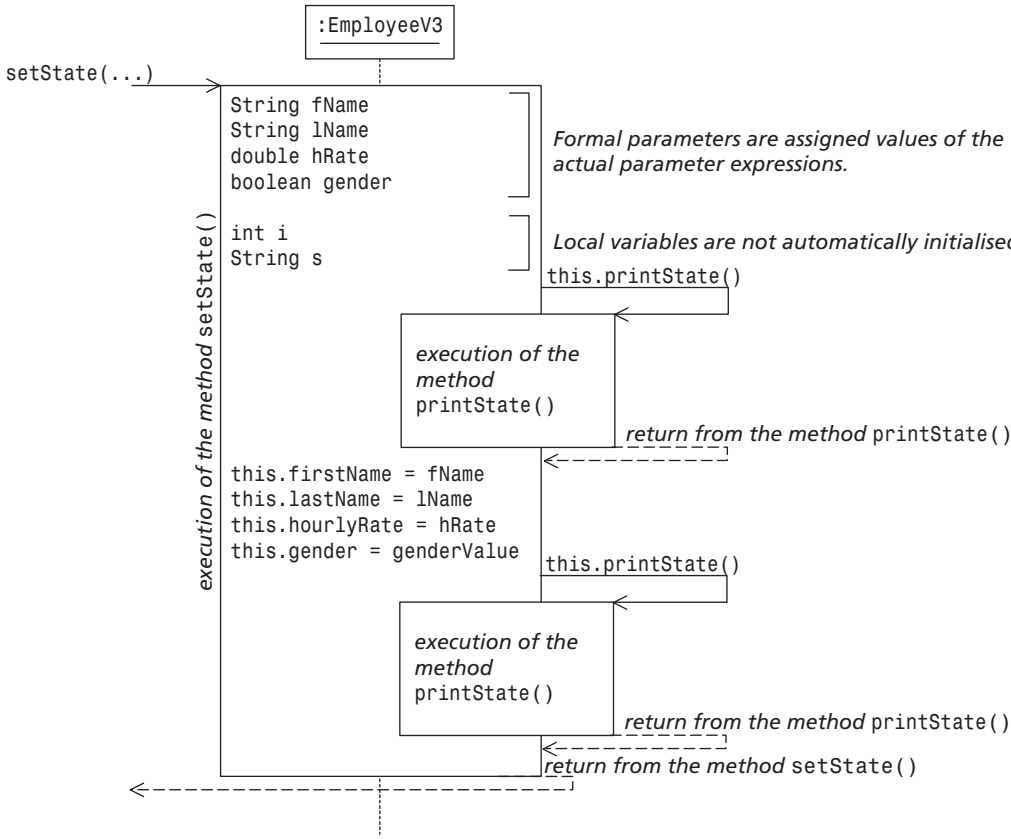
We can see from Figure 7.5 that control returns from the method `setState()` only after the last statement in the method has been executed. This need not always be the case: the `return` statement can be used to end execution of a method wherever appropriate.

The `return` statement comes in two forms. The first form of the statement consists of the keyword `return` only:

```
return;       // Method execution stops and control returns.
```

This form can be used when the method does not return a value, for example in the method `setState()`. We could have included a `return` statement as the last statement, but that would be redundant, as the execution of the method will end anyway when there are no more statements left to execute in the method body.

FIGURE 7.5 Method execution



The second form of the `return` statement requires an expression in addition to the keyword `return`. The value of this expression is returned after the execution of the method stops at the `return` statement. The method must explicitly specify the *return type* in the method header, and the *return value* must be of this return type. The `return` statement is required for methods that return a value, and the compiler will insist upon it. The method `computeSalary()` below illustrates the use of the `return` statement:

```
double computeSalary(double numOfHours) {
  assert numOfHours >= 0 : "Number of hours must be >= 0";
  double normalNumberOfHours = 37.5;
  double weeklySalary = hourlyRate * normalNumberOfHours;
  if (numOfHours <= normalNumberOfHours) {
    return weeklySalary;                                          // (1)
  }
  return weeklySalary +
         2.0 * hourlyRate * (numOfHours - normalNumberOfHours);   // (2)
}
```

The execution of the method stops when either (1) or (2) is executed. The method will return the value of the expression in the `return` statement that was executed. In both (1)

and (2) the expression evaluates to a value of type `double`, which is the return type specified in the method header. A method can return only one value, either a value of a primitive type or a reference value of an object.

The method `computeSalary()` above has two exits, corresponding to the two `return` statements. Execution can be difficult to understand if a method has too many exits. We can rewrite the method `computeSalary()` so that it has only one exit and in which the variable `weeklySalary` always holds the correct salary, allowing for overtime:

```
double computeSalary(double numOfHours) {
  assert numOfHours >= 0 : "Number of hours must be >= 0";
  double normalNumberOfHours = 37.5;
  double weeklySalary = hourlyRate * normalNumberOfHours;
  if (numOfHours > normalNumberOfHours) {
    weeklySalary += 2.0 * hourlyRate * (numOfHours - normalNumberOfHours);
  }
  return weeklySalary;
}
```

BEST PRACTICE

Keeping the number of exits from a method to a minimum aids in understanding the program logic.

Passing information using arrays

Program 7.4 shows two ways of passing information between methods using arrays.

At (1a) the method `main()` calls the method `fillStringArray()` declared at (3a), passing an empty array of strings. The method `fillStringArray()` reads as many strings as the length of the array, and fills the array. After control returns from the method `fillStringArray()`, the actual parameter `firstNameArray` still refers to the same array whose reference value was passed, but which is now filled with string values. The method call and the method declaration are as follows:

```
fillStringArray(firstNameArray);                          // (1a) Call
...
static void fillStringArray(String[] strArray) { ... } // (3a) Declaration
```

At (2a) the method `main()` calls the method `createStringArray()` declared at (4a). The method `createStringArray()` first asks the user for the number of values to read. It then creates an array of strings of this size, and subsequently fills the array. The method `createStringArray()` returns the reference value of the array, which on return is assigned to the local reference variable `lastNameArray`. The array object created in the method `createStringArray()` continues to exist even after the method has finished executing, because

the reference value of the array was stored in a local reference variable on return. The method call and the method declaration in this case are as follows:

```
String[] lastNameArray = createStringArray();// (2a) variable type String[]
...
static String[] createStringArray() { ... }  // (4a) return type String[]
```

Which approach is best depends on how much work the calling method wants the called method to do, i.e. more than just read the values and fill the array.

PROGRAM 7.4 Handling arrays

```
import java.util.Scanner;
public class ArrayMaker {

  public static void main(String[] args) {
    // (1) Read first names:
    String[] firstNameArray = new String[3];
    System.out.println("Read first names");
    fillStringArray(firstNameArray);                          // (1a)
    printStrArray(firstNameArray);

    // (2) Read last names:
    System.out.println("Read last names");
    String[] lastNameArray = createStringArray();             // (2a)
    printStrArray(lastNameArray);
  }

  // (3) Reference value of the array to be filled is passed
  //     as formal parameter:
  static void fillStringArray(String[] strArray) {            // (3a)
    Scanner keyboard = new Scanner(System.in);
    for (int i = 0; i < strArray.length; i++) {
      System.out.print("Next: ");
      strArray[i] = keyboard.nextLine();
    }
  }

  // (4) The method creates and fills an array of strings.
  //     The reference value of this array is returned by the method:
  static String[] createStringArray() {                       // (4a)
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter the number of items to read: ");
    int size = keyboard.nextInt();
    String[] strArray = new String[size];
    keyboard.nextLine();                    // Clear any input first
    for (int i = 0; i < strArray.length; i++) {
      System.out.print("Next: ");
      strArray[i] = keyboard.nextLine();
    }
```

```
    return strArray;
  }

  // (5) Prints the strings in an array to the terminal window:
  static void printStrArray(String[] strArray) {
    for (String str : strArray) {
      System.out.println(str);
    }
  }
}
```

Program output:

```
Read first names
Next: Tom
Next: Dick
Next: Linda
Tom
Dick
Linda
Read last names
Enter the number of items to read: 3
Next: Tanner
Next: Dickens
Next: Larsen
Tanner
Dickens
Larsen
```

Automatic garbage collection

Objects that are no longer in use are taken care of by the JVM without the program having to do anything special. These objects are deleted from memory, and the memory freed can be used for new objects. It is always the JVM that decides *when* such a clean up should take place, for example when it starts to run out of free memory. This clean-up process is called *automatic garbage collection*.

Note that if a method creates an object and returns its reference value, the reference value of the object can be used after the method returns. Such an object would therefore not be a candidate for garbage collection. If the reference value of an object is only stored in a local variable, the object will not be accessible after return from the method. Such an object can be garbage collected by the JVM.

7.4 Static members of a class

Static members specify properties and behaviour of the *class*, and are therefore not a part of any object created from the class.

The following simple example illustrates the use of static members. How can we keep track of the number of male and female employee objects that have been created? We can define two counters that are incremented, depending on whether a male or a female employee is created. If these counters are declared as instance variables in the class declaration, they will exist in every employee object we create from the class. Each object will have two counters, so which counters should we use to do the book keeping? One solution is to maintain the counters as static variables for the class only, and update the appropriate counter each time an employee object is created.

We use the keyword `static` in the declaration of static members to distinguish them from instance members. Program 7.5 shows declarations of static variables and methods at (1) and (2) respectively.

The class `EmployeeV4` in Program 7.5 declares the following two static variables (see under (1)):

```
static int numOfFemales;
static int numOfMales;
```

Static variables are *global variables* in the sense that there is only one occurrence of such variables and that is in the class, in contrast to instance variables that exist in each object created from the class.

Static variables are also useful for defining *global constants*. Once a value is assigned to such a variable its value cannot be changed. Such constants can be used inside the class and by other classes. If we decide that the Boolean value `true` indicates a female employee, then the Boolean value `false` indicates a male employee. Instead of using the Boolean values directly for this purpose, we can define and use constants with meaningful names (see (1) in Program 7.5):

```
static final boolean MALE = false;
static final boolean FEMALE = true;
```

The keyword `static` states that they can be considered as global variables, and the keyword `final` prevents the values assigned to these variables from being changed, i.e. they are global constants. Section 7.6 on page 192 illustrates a more elegant solution for defining a fixed number of such constants.

The class `EmployeeV4` also declares two static methods: `registerGender()` and `printStatistics()` at (2). The method `registerGender()` increments the relevant counter depending on the value of its parameter, and the method `printStatistics()` prints the number of male and female employees registered at any given time. We will put these methods to use in the next subsection.

Figure 7.6 shows the UML diagram for the class `EmployeeV4` with all its members. Static members are underlined in the diagram to distinguish them from instance members.

PROGRAM 7.5 Static members

```
class EmployeeV4 { // Assume that no constructors are declared.

  // (1) Static variables:
  static final boolean MALE = false;
  static final boolean FEMALE = true;
  static final double NORMAL_WORKWEEK = 37.5;
  static int numOfFemales;
  static int numOfMales;

  // (2) Static methods:
  // Register an employee's gender by updating the relevant counter.
  static void registerGender (boolean gender) {
    if (gender == FEMALE) {
      ++numOfFemales;
    } else {
      ++numOfMales;
    }
  }
  // Print statistics about the number of males and females registered.
  static void printStatistics() {
    System.out.println("Number of females registered: " +
                       EmployeeV4.numOfFemales);                    // (3)
    System.out.println("Number of males registered: " +
                       EmployeeV4.numOfMales);                      // (4)
  }

  // Rest of the specification is the same as in class EmployeeV3
  // ...

}
// Accessing static members
public class Client4A {

  public static void main(String[] args) {
    // (5) Print information in class EmployeeV4 before any objects
    //     are created:
    System.out.println("Print information in class EmployeeV4:");
    System.out.println("Females registered: " +
                       EmployeeV4.numOfFemales);        // (6) class name
    System.out.println("Males registered: " +
                       EmployeeV4.numOfMales);          // (7) class name

    // (8) Create a male employee.
    EmployeeV4 coffeeboy = new EmployeeV4();
    coffeeboy.setState("Tim", "Turner", 30.00, EmployeeV4.MALE);
    coffeeboy.registerGender(EmployeeV4.MALE);          // (9) referanse
    System.out.println("Print information in class EmployeeV4:");
```

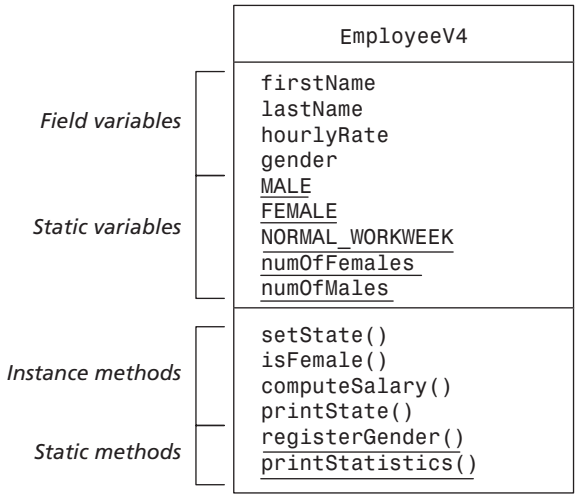

```
    coffeeboy.printStatistics();                          // (10) referanse

    // (11)  Create a female employee.
    EmployeeV4 receptionist = new EmployeeV4();
    receptionist.setState("Amy", "Archer", 20.50, EmployeeV4.FEMALE);
    EmployeeV4.registerGender(EmployeeV4.FEMALE);
    System.out.println("Print information in class EmployeeV4:");
    EmployeeV4.printStatistics();
  }
}
```

Program output:

```
Print information in class EmployeeV4:
Females registered: 0
Males registered: 0
Print information in class EmployeeV4:
Number of females registered: 0
Number of males registered: 1
Print information in class EmployeeV4:
Number of females registered: 1
Number of males registered: 1
```

FIGURE 7.6 UML diagram showing all members in a class



Accessing static members

A static member can be accessed using the notation *className.memberName*. At (3) and (4) in Program 7.5 the method `printStatistics()` refers to the static variables `numOfFemales` and `numOfMales` using this notation: `EmployeeV4.numOfFemales` and `EmployeeV4.numOfMales`.

We can use the member name to refer to static members in the same class inside any method, provided that the name is not shadowed by a local variable. The method `registerGender()` in Program 7.5 refers to the static variables `numOfFemales` and `numOfMales` in class `EmployeeV4`.

Inside an instance method, we can also use the `this` reference to refer to static members in the same class. The `this` reference refers to the current object, and an object has a reference type (for example, its class), thus the notation `this.staticMember` uniquely identifies the member in the class.

In Program 7.1 on page 166 the method `computeSalary()` in the class `EmployeeV3` declares a local variable `normalNumOfHours` that represents the normal number of working hours in a week (37.5). If other methods need this information, this value must be duplicated locally where it is needed. If the normal number of hours changes in the future, the value must be changed in all places in the code where it is used. This value is an obvious candidate for a global constant:

```
static final double NORMAL_WORKWEEK = 37.5;
```

Now it is only necessary to change the value in one place and recompile the code. Its name makes it obvious what the value represents. The method `computeSalary()` below uses the value of the global constant `NORMAL_WORKWEEK`. This new declaration of the method shows the three ways in which we can refer to a static member inside an instance method in the same class:

```
double computeSalary(double numOfHours) {
  assert numOfHours >= 0 : "Number of hours must be >= 0";
  double weeklySalary = hourlyRate * this.NORMAL_WORKWEEK;             // (1)
  if (numOfHours > EmployeeV4.NORMAL_WORKWEEK) {                       // (2)
    weeklySalary += 2.0 * hourlyRate * (numOfHours - NORMAL_WORKWEEK); // (3)
  }
  return weeklySalary;
}
```

Program 7.5 also shows how static members of a class can be accessed from other classes using the class name, as shown at (6) and (7). They can also be accessed using a reference of the class. The method `main()` in class `Client4A` calls the static methods `registerGender()` and `printStatistics()` in class `EmployeeV4`, using the reference `coffeeboy` at (9) and (10). We don't need to create an object of the class to refer to the static members of the class.

No `this` reference for static members

Static methods cannot refer to instance members by name, not even by using the `this` reference – static methods have no `this` reference. They are a part of the class, and not a part of the current object.

Initializing static variables

Static members of a class are initialized automatically before the class is used. If a static variable specifies an initial value, the static variable will be initialized with that value, otherwise it will be initialized with the default value of its data type specified in the declaration.

Output from (6) and (7) in Program 7.5 correctly shows that the static variables `EmployeeV4.numOfFemales` and `EmployeeV4.numOfMales` are initialized to the default value 0.

BEST PRACTICE

To aid in understanding the program, it is a good idea to initialize static variables in their declaration, and to use the class name when accessing static members.

The `main()` method and program arguments

A Java application must have a primary class that defines a method `main` and which has the method header:

```
public static void main(String[] args)
```

Execution always starts in the `main()` method of the class. In the examples we have seen so far, the program terminates when all the actions in the `main()` method have been executed.

The signature of the `main()` method above shows that it has an array of strings (`String[]`) as a formal parameter. *Program arguments* specified on the command line are stored in an array of strings. The reference value of this array is assigned to the formal parameter `args` before the execution of the `main()` method starts. In the program, the `main()` method can obtain these program arguments from the array using the formal parameter variable `args`. The strings in the array can be indexed in the usual way, with the first string as `args[0]` and the last string as `args[args.length-1]`. If there are no program arguments, the parameter variable `args` will refer to an array of strings with zero elements, i.e. `args.length` is zero. Any attempt to index a value in this case will result in an `ArrayIndexOutOfBoundsException`, as the array has no elements.

Program 7.6 shows an example in which the program receives information about an employee from the command line in the form of program arguments. The command line specifies the following information:

```
> java Client4B Mona Lisa 20.5 true
```

The program arguments are specified after the class name. Program 7.6 checks at (1) that the array `args` has exactly four program arguments. If this is the case, the program arguments are printed at (2). The values in the `args` array are assigned to local variables at (3). At (4) and (5) the string values are converted to values of the correct type. An employee object is created and assigned these values at (6). Note that the values from the command line are passed as strings, and must be explicitly converted to other values if it is necessary in the program.

PROGRAM 7.6 Reading program arguments

```
// Using program arguments
public class Client4B {
```

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

    // (1) Check that all information about an employee is given
    //     on the command line:
    if (args.length != 4) {
      return;
    }

   // (2) Print the array args:
    System.out.println("Program arguments:");
    for (String arg : args) {
      System.out.println(arg);
    }

    // (3) Assign information from the array args to local variables:
    String firstName = args[0];
    String lastName = args[1];
    double hourlyRate = Double.parseDouble(args[2]); // (4) Floating-point
    boolean gender;
    if (args[3].equals("true")) {                    // (5) Boolean value
      gender = EmployeeV4.FEMALE;
    } else {
      gender = EmployeeV4.MALE;
    }

    // (6) Create an employee, and print its state:
    EmployeeV4 decorator = new EmployeeV4();
    decorator.setState(firstName, lastName, hourlyRate, gender);
    System.out.println("Information about an employee:");
    decorator.printState();
    System.out.printf("Salary: %.2f%n", decorator.computeSalary(40.0));
  }
}
```

Program output:

```
> javac EmployeeV4.java Client4B.java
> java Client4B Mona Lisa 20.5 true
Program arguments:
Mona
Lisa
20.5
true
Information about an employee:
First name: Mona   Last name: Lisa   Hourly rate: 20.50   Gender: Female
Salary: 871.25
```

7.5 Initializing object state

Default constructors: implicit or explicit

Constructors have a special role in a class declaration. The use of the `new` operator, together with a constructor call, results in the execution of the constructor that corresponds to the constructor call. The main purpose of a constructor is to set the initial state of the current object, i.e. the object that has been created by the `new` operator.

If the class `EmployeeV5` does not declare a constructor, the compiler will generate a constructor for the class equivalent to the following declaration:

```
EmployeeV5() { ... }    // (1)
```

Note the use of the class name and the absence of parameters in the constructor header. The actual contents of the constructor body are not important for this discussion, except to note that the constructor body has *no* actions to initialize the state of the current object.

The constructor declaration resembles a method declaration, but it is not a method. It is called the *default constructor* for the class `EmployeeV5`. Because it was generated by the compiler, it is called the *implicit default constructor*. If we create an object of this class:

```
EmployeeV5 cook = new EmployeeV5();   // (2)
```

the constructor call `EmployeeV5()` will result in the constructor at (1) being executed. This constructor has no effect on the state of the current object.

As we can see, the implicit default constructor is not always adequate, as it does *not* affect the state of the current object. The field variables will always be initialized to their default values (or to any initial value specified in the field declarations). Therefore a class will usually provide explicit constructors to set the initial state of an object. This ensures that the object is initialized properly before it is used.

A class can choose to declare an *explicit default constructor*. For example, the class `Employee5V` declares such a constructor at (3):

```
class EmployeeV5 {
  // Explicit default constructor
  EmployeeV5() {                              // (3) No parameters
    firstName  = "Joe";
    lastName   = "Jones";
    hourlyRate = 15.50;
    gender     = MALE;
  }
  // Rest of the specification is the same as in class EmployeeV4
}
```

Now the constructor call at (2) above will result in the explicit default constructor at (3) being executed. Each object created in this way will always have the same state, as shown in Figure 7.2b on page 164. A standard constructor is not adequate, however, for situations that require a more customised initial object state.

A constructor always has the same name as the class, so that the signature comprises the class name and type of the formal parameters. It is called in conjunction with the `new` operator to return the reference value of the object that has been created. A constructor cannot return a value. Apart from that, the constructor body can contain declarations and actions, similar to an instance method body. A constructor can use the `this` reference to refer to the current object, and all members in the class can be accessed in the constructor body.

Constructors with parameters

A class can declare constructors with formal parameters, to create objects with appropriate initial states. Constructors with parameters are called *non-default constructors*.

Program 7.7 shows the use of non-default constructors to initialize fields in objects created with the `new` operator. Declarations (2) and (3) create two objects of the class `EmployeeV6`. Parameter passing takes place in the same way as for method calls. The initial state of the objects referred to by the references `operator1` and `operator2` are shown in Figure 7.7a and Figure 7.7b respectively. In addition to initializing the state of the object with the values of the actual parameter expressions in the constructor call, the constructor in the class `EmployeeV6` also calls the static method `registerGender()` to update the count of male and female employees.

If a class declares *any* constructor, the *implicit* default constructor cannot be applied. In Program 7.7, the declaration at (1) results in a compiler error:

```
EmployeeV6 clerk = new EmployeeV6();   // (1) Compile-time error!
```

The signature of the constructor call at (1) is not compatible with the signature of the non-default constructor:

```
EmployeeV6(String, String, double, boolean)  // Non-default constructor
                                             // signature
```

The class `EmployeeV6` must declare the default constructor *explicitly* to allow this constructor to be called.

> **BEST PRACTICE**
>
> Always declare an explicit default constructor.

FIGURE 7.7 Initializing of object state with non-default constructors

```
EmployeeV6 operator1 = new EmployeeV6("Tim", "Tanner", 20.60, EmployeeV6.MALE);   // (2)
EmployeeV6 operator2 = new EmployeeV6("Amy", "Archer", 18.50, EmployeeV6.FEMALE); // (3)
```

(a) (b)

PROGRAM 7.7 Non-default constructors

```
class EmployeeV6 {
  // Only non-default constructor
  EmployeeV6(String fName, String lName, double hRate, boolean gender) {
    this.firstName = fName;           // field access via this reference
    this.lastName = lName;
    this.hourlyrate = hRate;
    this.gender = gender;
    this.registerGender(gender);      // call to static method
  }
  // Rest of the specification is the same as in class EmployeeV4
  // ...
}
// Using constructors
public class Client6 {

  public static void main(String[] args) {

    // EmployeeV6 clerk = new EmployeeV6();     // (1) Compile-time error!
                                                // No default constructor.
    // Print information in class EmployeeV6
    System.out.println("Print information in class EmployeeV6:");
    EmployeeV6.printStatistics();
    System.out.println();

    // Create an employee, and print its information
    EmployeeV6 operator1 = new EmployeeV6("Tim", "Turner", 30.00,
                                          EmployeeV6.MALE);          // (2)
    printEmployeeInfo(operator1, 40.0);
    System.out.println();

    // Create a new employee, and print its information
    EmployeeV6 operator2 = new EmployeeV6("Amy", "Archer", 20.50,
                                        EmployeeV6.FEMALE);          // (3)
    printEmployeeInfo(operator2, 50.0);
  }
```

```
  static void printEmployeeInfo(EmployeeV6 employee,
                                double numOfHours) {
    System.out.println("Printing information about an employee:");
    employee.printState();
    System.out.printf("Salary: %.2f%n",
                      employee.computeSalary(numOfHours));
    System.out.println("Print information in class EmployeeV6:");
    EmployeeV6.printStatistics();
  }
}
```

Program output:

```
Print information in class EmployeeV6:
Number of females registered: 0
Number of males registered: 0

Printing information about an employee:
First name: Tim Last name: Turner   Hourly rate: 30.00  Gender: Male
Salary: 1275.00
Print information in class EmployeeV6:
Number of females registered: 0
Number of males registered: 1

Printing information about an employee:
First name: Amy Last name: Archer   Hourly rate: 20.50  Gender: Female
Salary: 1281.25
Print information in class EmployeeV6:
Number of females registered: 1
Number of males registered: 1
```

Overloading constructors

If necessary a class can declare several constructors. Each constructor call will result in the execution of the constructor that has a signature compatible with the constructor call, analogous to method calls – see *Method calls and actual parameter expressions* on page 167.

Program 7.8 shows a class `EmployeeV7` that declares three constructors. We say that the constructors are *overloaded*, i.e. they have the same class name, but their signatures differ because they have different formal parameter lists.

The class `Client7` creates three objects of the `EmployeeV7` class and prints pertinent information about them. The initial state of each object is dependent on which constructor was executed when the object was created. Creating the object at (4) results in the constructor at (1) being executed. Creating the object at (5) results in the constructor at (2) being executed. Finally, creating the object at (6) results in the constructor at (3) being executed.

BEST PRACTICE

Any initialization code in the constructor overrides initial values specified in the field declarations. Therefore it is a good idea always to initialize fields in one place: in the constructor.

PROGRAM 7.8 Constructor overloading

```
class EmployeeV7 {
  static final double STANDARD_HOURLY_RATE = 15.50;

  // Constructors
  EmployeeV7() {                                                    // (1)
    firstName  = "Joe";
    lastName   = "Jones";
    hourlyRate = STANDARD_HOURLY_RATE;
    gender     = MALE;
    registerGender(MALE);
  }

  EmployeeV7(String fName, String lName, boolean gender) {     // (2)
    this.firstName  = fName;
    this.lastName   = lName;
    this.hourlyRate = STANDARD_HOURLY_RATE;
    this.gender     = gender;
    this.registerGender(gender);
  }

  EmployeeV7(String fName, String lName,
             double hRate, boolean gender) {                        // (3)
    this.firstName  = fName;
    this.lastName   = lName;
    this.hourlyRate = hRate;
    this.gender     = gender;
    this.registerGender(gender);
  }

  // The rest of the specification is the same as in class EmployeeV6
}
// Using overloaded constructors
public class Client7 {

  public static void main(String[] args) {

    // Print information in class EmployeeV7
    System.out.println("Printing information in class EmployeeV7:");
    EmployeeV7.printStatistics();
```

```
    // Create an employee, and print its information
    EmployeeV7 guard1 = new EmployeeV7();                          // (4)
    printEmployeeInfo(guard1, 50);

    // Create an employee, and print its information
    EmployeeV7 guard2 = new EmployeeV7("Tim", "Turner",
                                       EmployeeV7.MALE);           // (5)
    printEmployeeInfo(guard2, 40);

    // Create a new employee, and print its information
    EmployeeV7 guard3 = new EmployeeV7("Amy", "Archer", 20.50,
                                       EmployeeV7.FEMALE);         // (6)
    printEmployeeInfo(guard3, 35);

    // Print information in class EmployeeV7
    System.out.println("\nPrinting information in class EmployeeV7:");
    EmployeeV7.printStatistics();
  }

  static void printEmployeeInfo(EmployeeV7 employee, double numOfHours) {
    System.out.println();
    System.out.println("Printing information about an employee:");
    employee.printState();
    System.out.printf("Salary: %.2f%n",
                      employee.computeSalary(numOfHours));
  }
}
```

Program output:

```
Printing information in class EmployeeV7:
Number of females registered: 0
Number of males registered: 0

Printing information about an employee:
First name: Joe Last name: Jones    Hourly rate: 15.50  Gender: Male
Salary: 968.75

Printing information about an employee:
First name: Tim Last name: Turner   Hourly rate: 15.50  Gender: Male
Salary: 658.75

Printing information about an employee:
First name: Amy Last name: Archer   Hourly rate: 20.50  Gender: Female
Salary: 768.75

Printing information in class EmployeeV7:
Number of females registered: 1
Number of males registered: 2
```

7.6 Enumerated types

Simple form of enumerated types

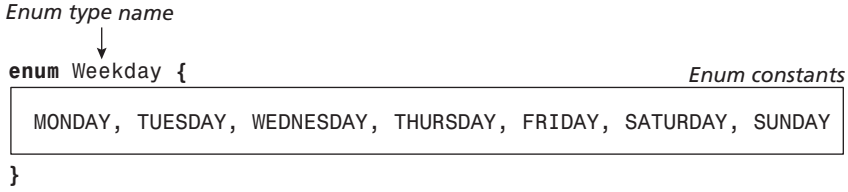
An *enumerated type* (also called *enum* for short) defines a fixed number of enum constants. An enumerated constant is a unique name that refers to a particular object. Figure 7.8 shows the simple form of the enum type. The keyword `enum` indicates that the declaration is an enumerated type. The enum constants are specified in a list in the block that comprises the body of the enumerated type. Thus the enum type `Weekday` defines seven enum constants, one for each day of the week.

The set of objects for an enum type is confined to the objects listed by the declaration of the enum type. These objects are created automatically only once during program execution. It is not possible to create more objects of the enum type with the `new` operator. We can consider the enum constants as global constants that are declared `static` and `final`.

The enum type `Weekday` defines a reference type. Just as with other reference types, we can declare variables of the enum type, but these variables can only refer to objects that are designated by the enum constants:

```
Weekday lateOpeningDay = Weekday.THURSDAY;
```

FIGURE 7.8 Simple form of enumerated types



Selected methods for enumerated types

Table 7.1 shows two methods that can be applied to all enumerated types. The method `toString()` returns the name of the enum constant. This method is applied implicitly to the parameters of the `println()` method in the following code examples:

```
System.out.println(lateOpeningDay);          // Prints THURSDAY.
System.out.println(Weekday.SUNDAY);          // Prints SUNDAY.
```

We can also compare constants in an enumerated type:

```
assert(lateOpeningDay != Weekday.SUNDAY);    // true
assert(lateOpeningDay == Weekday.THURSDAY); // true
```

Enum constants can also be used as `case` labels in a `switch` statement. Given that the variable `day` has the type `Weekday`, the `switch` statement below will determine whether it is a working day or falls on a weekend. Note that enum constants in `case` labels are not

referred to by the name of the enum type, because it is implied by the type of the `switch` statement, which in this case is the type of the variable `day`, i.e. `Weekday`:

```
switch(day) {                                // (1)
  case SATURDAY: case SUNDAY:
    System.out.println("The day is " + day + ", it must be weekend.");
    break;
  default:
    System.out.println(day + " is a working day.");
}
```

If we need to iterate over all the constants of an enumerated type, we can first create an array with all the constants by calling the static method `values()`, and then use a `for(:)` loop to iterate over the array:

```
Weekday[] daysArray = Weekday.values();
for (Weekday day : daysArray) {
  // switch statement given at (1) above
}
```

Program 7.9 shows the execution of the code examples presented in this subsection.

TABLE 7.1 Selected method for all enumerated types

Method	Description
`String toString()`	Returns the string representation of the current enum constant, i.e. the name of the current constant.
`static` *`enumTypeName`*`[] values()`	Returns an array with the enum constants that are declared in the enum type that has the *enumTypeName*. The order of the constants in the array is the same as the order in the enum declaration.

PROGRAM 7.9 Use of enumerated types

```
// An enum client
public class Weekdays {
  public static void main(String[] args) {

    Weekday lateOpeningDay = Weekday.THURSDAY;  // Reference of enum type

    // Method toString() applied implicitly
    System.out.println(lateOpeningDay);         // Prints THURSDAY.
    System.out.println(Weekday.SUNDAY);         // Prints SUNDAY.

    // Testing for equality
    assert(lateOpeningDay != Weekday.SUNDAY);   // true
    assert(lateOpeningDay == Weekday.THURSDAY); // true
```


```
    // Iterate over days of the week:
    System.out.println("Days of the week:");
    Weekday[] daysArray = Weekday.values();
    for (Weekday day : daysArray) {
      switch(day) {                              // (1)
        case SATURDAY: case SUNDAY:
          System.out.println("The day is " + day +
                             ", it must be weekend.");
          break;
        default:
          System.out.println(day + " is a working day.");
      }
    }
  }
}
```

Program output:

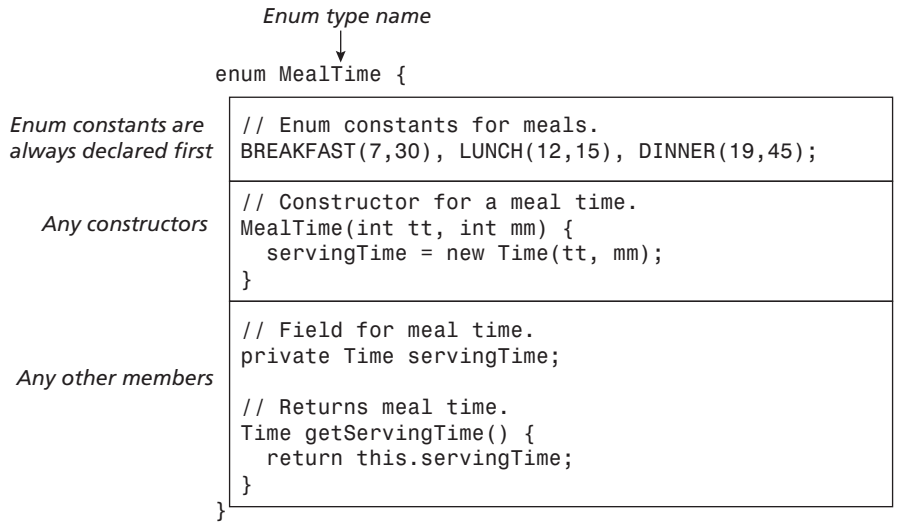
```
THURSDAY
SUNDAY
Days of the week:
MONDAY is a working day.
TUESDAY is a working day.
WEDNESDAY is a working day.
THURSDAY is a working day.
FRIDAY is a working day.
The day is SATURDAY, it must be weekend.
The day is SUNDAY, it must be weekend.
```

General form of enumerated types

The general form of the enumerated type declaration is shown in Figure 7.9. An enumerated type can declare constructors and other members, analogous to those in a class declaration. In this way, it is possible to define properties and behaviour of enum constants in an enumerated type.

Constructors cannot be called directly. A constructor is called *implicitly* when an object representing an enum constant is created automatically from the enum declaration during execution. Each constant name in the enum type `MealTime` is declared with an *actual* parameter list. The constructor which has the corresponding formal parameter list is executed. Specifying `BREAKFAST(7, 30)` in the enum constant list results in the constructor being called with `7` and `30` as values of the formal parameters `tt` and `mm`, representing hours and minutes, respectively.

FIGURE 7.9 The general form of enumerated types



The enumerated type `MealTime` in Figure 7.9 declares a field, `servingTime`, of reference type `Time` (see the listing in Program 7.9). Each object of the enumerated type `MealTime` will have this field, which will be initialized by the call to the constructor. The three objects, corresponding to each of the enum constants, will have states corresponding to the serving times for the different meals. The enumerated type `MealTime` also declares an instance method, `getServingTime()`, which returns the time for serving a meal. The enumerated type `MealTime` has only three objects that are referred to by the enum constants in the declaration. We can invoke the method `getServingTime()` on these objects in the usual way:

```
System.out.println(MealTime.BREAKFAST.getServingTime()); // Prints 07:30
```

The class `MealService` in Program 7.9 prints the times when the different meals are served.

PROGRAM 7.10 Serving meals

```
// Time is given as hours (0-23) and minutes (0-59).
class Time {

  // Fields for the time.
  int hours;
  int minutes;

  // Constructor
  Time(int hours, int minutes) {
    assert (0 <= hours && hours <= 23 &&
            0 <= minutes && minutes <= 59) :
            "Invalid hours and/or minutes";
    this.hours = hours;
```


```
    this.minutes = minutes;
  }

  // String representation of the time, TT:MM
  public String toString() {
    return String.format("%02d:%02d", hours, minutes);
  }
}
// Using enums
public class MealService {
  public static void main(String[] args) {

    // (1) Create an array of meals:
    MealTime[] meals = MealTime.values();

    // (2) Print meal times:
    for (MealTime meal : meals) {
      System.out.println(meal + " is served at " + meal.getServingTime());
    }
  }
}
```

Program output.

```
> javac MealTime.java Time.java MealService.java
> java -ea MealService
BREAKFAST is served at 07:30
LUNCH is served at 12:15
DINNER is served at 19:45
```

Declaring enumerated types inside a class

So far we have declared an enumerated type as a top-level declaration in its own separate source file. Other clients can access enum constants by using the class name and the constant name. An enumerated type can also be declared as a member in a class declaration. It makes sense to do this if the use of the enum constants is localised to a single class. The code below shows the declaration of the enumerated type `Weekday` as a member of the class `Weekdays`. Access to the enum constants is the same as it was before (see also Program 7.9):

```
// Enum type as member in a class
public class Weekdays {
  // Enum type as member in a class.
  enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
  }
  public static void main(String[] args) {
    // Same as before.
  }
}
```

7.7 Review questions

1. Identify all the members and constructors in the declaration of the class `Counter`. Group them as shown in Figure 7.1.

```
class Counter {
  final static int MAX_VALUE = 100;
  static String description = "This class creates counters.";
  int value;

  Counter() { value = 1; }
  Counter(int initialValue) { value = initialValue; }

  int getCounter() { return value; }
  void setCounter(int newValue) { value = newValue; }
  void incrementCounter() { ++value; }
  void decrementCounter() { --value; }
  void resetCounter() { value = 0; }

  static String getDescription() { return description; }
}
```

2. _____ belong to objects, while _____ belong to the class.

3. Values of all the field variables in an object comprise its _____.

4. What do we mean by the initial state of an object?

5. What is the default value for any reference variable?

6. In the class `RectangleV2`, which field variables are initialized with default values and which are initialized with an initial value when an object of the class is created?

```
class RectangleV2 {
  double length;
  double breadth;
  double area = length * breadth;
}
```

7. Which of the following statements is true?

   a. Members in a class can be declared in any order.
   b. A class must always provide an explicit default constructor.
   c. The implicit default constructor is always executed when an object of the class is created using the `new` operator.
   d. The explicit default constructor can have formal parameters.

8. In the class `ClientA`, identify the class names, local variables, method calls, formal parameters and actual parameters. The class `Counter` is defined in Question 1.

```
// Counting cars
public class ClientA {
  public static void main(String[] args) {
    int numOfCars = 10;
    Counter carCounter = new Counter(numOfCars);
    carCounter.incrementCounter();
    System.out.println(carCounter.getCounter());
  }
}
```

9. Given the class `Variables` and a reference `obj` that refers to an object of this class, what will be printed by the method call `obj.doIt(2)`?

```
class Variables {
  static String s = " is equal to ";

  double d = 20;

  int k = 10;

  void doIt(int d) {
    for (int k = 0; k < 2; k++) {
      System.out.println("k + d" + s + (k + d)); // (1)
    }
    System.out.println("k + d" + s + (k + d));   // (2)
    String s = " = ";
    System.out.println("k + d" + s + (k + d));   // (3)
  }
}
```

10. Which statements are true about methods?

- **a** The method signature comprises the return type, method name and type of the formal parameters.
- **b** Formal and actual parameters must be compatible with regard to type, number and order.
- **c** Parameter passing in Java entails that the values of the actual parameter expressions are assigned to the corresponding formal parameter variables.
- **d** A method always returns a value.
- **e** The return value must always be assigned to a variable.

11. Answer "yes" or "no" to the following questions:

- **a** Is it possible to declare several methods with the same name, but different parameter lists, in the same class?
- **b** If the actual parameter in a method call is a variable name, can the corresponding formal parameter variable have the same name?
- **c** Can a `void` method contain a `return` statement?

12. Determine whether you would use a `void` or a non-`void` method for the following tasks. If implementing a non-`void` method, which return type would you choose?

- **a** Determine whether a person has reached retirement age.
- **b** Give an explanation to the user of what the program does.
- **c** Find the largest of two numbers.
- **d** Read an `int` value entered by the user.
- **e** Assign an `int` value to a field variable.
- **f** Create an array of `Counter` objects for a client (see Question 1).
- **g** Create a two-dimensional array of strings.
- **h** Create statistics for the number of newspapers sold per day for a four-week period, given that the weekdays are numbered from 0 upwards (Monday has weekday number 0).

13. What will the following program print? The class `Counter` is declared in Question 1.

```
// Parameter passing
public class ParameterClient {
  public static void main(String[] args) {
    Counter counter = new Counter();
    int numOfTimes = 5;

    System.out.println("Before method call: " +
        "counter value is equal to " + counter.getCounter() +
        " and number of times is " + numOfTimes);
    updateCounter(counter, numOfTimes);
    System.out.println("After method call: " +
        "counter value is equal to " + counter.getCounter() +
        " and number of times is " + numOfTimes);
  }

  static void updateCounter(Counter counter, int numOfTimes) {
    for (; numOfTimes > 0; --numOfTimes) {
      counter.incrementCounter();
    }
  }
}
```

14. What will the following program print? The class `Counter` is declared in Question 1.

```
// More parameter passing
public class SwapClient {
  public static void main(String[] args) {
    int startvalue = 10;
    Counter counter1 = new Counter(startvalue);
    Counter counter2 = new Counter(startvalue * 2);
    System.out.println("Before swapping: " +
        "counter1 is " + counter1.getCounter() +
        " and counter2" + " is " + counter2.getCounter());
    swap(counter1, counter2);
    System.out.println("After swapping: " +
```


```
            "counter1 is " + counter1.getCounter() +
            " and counter2" + " is " + counter2.getCounter());
    }

    static void swap(Counter counter1, Counter counter2) {
      Counter t3 = counter1;
      counter1 = counter2;
      counter2 = t3;
    }
}
```

15. Given the following declarations, where the class `Counter` is declared as in Question 1:

```
Counter counterA = new Counter();
Counter[] counterArrayA = new Counter[5];
Counter[][] counterArrayB = new Counter[2][3];
```

and the following method declaration:

```
static void playingWithParameters(Counter counter, Counter[] array)
    { /* ... */ }
```

Which of these method calls are valid?

a `playingWithParameters(counterA, counterArrayA[]);`

b `playingWithParameters(counterA, counterArrayA);`

c `playingWithParameters(counterArrayA[2], counterArrayA);`

d `playingWithParameters(counterArrayA[3], counterArrayB[0]);`

e `playingWithParameters(counterArrayB[1][3], counterArrayA);`

f `playingWithParameters(counterArrayB[0][2], counterArrayB[1]);`

g `playingWithParameters(counterA, counterArrayB);`

16. Rewrite the class `Counter` declared in Question 1 using the `this` reference explicitly.

17. Which of these statements are valid, given the declaration of the class `Counter` from Question 1, and that `ref` refers to an object of this class?

a `System.out.println(ref.value);`

b `System.out.println(Counter.value);`

c `System.out.println(ref.MAX_VALUE);`

d `System.out.println(Counter.MAX_VALUE);`

e `System.out.println(ref.getCounter());`

f `System.out.println(Counter.getCounter());`

g `System.out.println(ref.getDescription());`

h
```
System.out.println(Counter.getDescription());
```

18. Which of the following statements are true about constructors?

a A constructor is a method.

b A constructor must specify a return type.

c If a class declares more than one constructor, then these are overloaded.

d Actions in a constructor can refer to all members of the class.

e The `this` reference cannot be used in a constructor.

19. Given the following class:

```
// Being four-sided
public class FourSided {
  public static void main(String[] args) {
    Rectangle shape = new Rectangle();
    System.out.println(shape.length * shape.breadth);
  }
}
```

What will the program above print if we use the following declarations for the class `Rectangle`?

a
```
class Rectangle {
  double length;
  double breadth;
}
```

b
```
class Rectangle {
  double length = 20;
  double breadth = 30;
}
```

c
```
class Rectangle {
  double length = 20;
  double breadth = 30;

  Rectangle() {
    length = 10;
    breadth = 5;
  }
}
```

20. Given the following variable declarations (1) and (2):

```
Counter counter1 = new Counter();  // (1)
Counter counter2 = new Counter(2); // (2)
```

Determine whether the variable declarations (1) and (2) are valid if we use the following declarations of the class `Counter`:

a
```
class Counter {
  int value;
```

```
}
```

b
```
class Counter {
  int value;
  Counter() { value = 1; }
}
```

c
```
class Counter {
  int value;
  Counter(int startValue) { value = startValue; }
}
```

d
```
class Counter {
  int value;
  Counter() { value = 1; }
  Counter(int startValue) { value = startValue; }
}
```

21. Which lines of code in the declaration of the `NewCounter` class will result in a compile- time error?

```
class NewCounter {
  static String description = "A new class that creates counters.";

  int value;

  NewCounter(int initialValue) {                  // A constructor
    System.out.println(value);                   // (1)
    System.out.println(this.value);              // (2)
    System.out.println(NewCounter.value);        // (3)
    System.out.println(description);             // (4)
    System.out.println(this.description);        // (5)
    System.out.println(NewCounter.description); // (6)
    value = initialValue;
  }

  void resetCounter() {                          // An instance method
    NewCounter t1 = new NewCounter(10);
    System.out.println(value);                   // (7)
    System.out.println(this.value);              // (8)
    System.out.println(t1.value);                // (9)
    System.out.println(NewCounter.value);        // (10)
    System.out.println(description);             // (11)
    System.out.println(this.description);        // (12)
    System.out.println(t1.description);          // (13)
    System.out.println(NewCounter.description); // (14)
    value = 0;
  }

  static String getDescription() {               // A static method
    NewCounter t1 = new NewCounter(10);
    System.out.println(value);                   // (15)
```

```
        System.out.println(this.value);              // (16)
        System.out.println(t1.value);                // (17)
        System.out.println(NewCounter.value);        // (18)
        System.out.println(description);             // (19)
        System.out.println(this.description);        // (20)
        System.out.println(t1.description);          // (21)
        System.out.println(NewCounter.description); // (22)
        return description;
    }
}
```

22. Which of the following statements are not true of enumerated types?

a An enumerated type is a reference type defined with the keyword `enum`.

b An enumerated type can be declared in its own separate source code file.

c An enumerated type can declare constructors.

d An enumerated type can declare members, as in a class declaration.

e Enum constants must always be declared first in an enumerated type declaration.

f We can use the `new` operator to create objects of an enumerated type.

23. Given the following declaration of an enumerated type:

```
enum LightColour { RED, YELLOW, GREEN }
```

and the following variable declaration:

```
LightColour colour;
```

Which of these statements will not compile?

a
```
colour = GREEN;
```

b
```
for (LightColour colour : LightColour) {
  System.out.println(colour);
}
```

c
```
switch(colour)
  case LightColour.RED: System.out.println("STOP!"); break;
  case LightColour.GREEN: System.out.println("GO!"); break;
  case LightColour.YELLOW: System.out.println("CAREFUL!"); break;
  default: assert false: "UNKNOWN COLOUR!";
}
```

d
```
Boolean b = colour.equals(GREEN);
```

e
```
LightColour.GREEN = colour;
```

7.8 Programming exercises

1. Modify the class `Counter` from Question 1 on page 197 so that it meets the following criteria:

- It is possible to create a counter that counts within an interval. The interval is given by a lower and an upper limit. It should be possible to specify the initial value of the counter, which must be within the interval.
- If no initial value is specified together with the interval, counting starts from the lower limit of the interval.
- If no interval is specified, but the initial value *is* specified, counting starts with the initial value and continues upwards.
- If neither the interval nor the initial value are specified, counting starts at 0 and continues upwards.
- It should be possible to increment and decrement a counter.
- It should be possible to get the current value of a counter, and to reset a counter to an initial value.
- Where necessary, check that the counter has a valid value.

Write a separate class to test the class `Counter`.

2. We will write a class called `Forex` that can be used for converting between two currencies. An object of the class stores the exchange rate between two currencies (for example, NOK 11.54 = GBP 1). The class offers two methods for converting between the two currencies. Choose suitable names for all members.

Write a client that creates several `Forex` objects and tests conversion between different currencies.

3. Write a class `Reverse` that reads program arguments and prints them in reverse. In the printout the arguments should be separated by comma (`,`). Write a separate method that takes care of the printout.

Example of program execution:

```
> java Reverse To be or not to be
be, to, not, or, be, To
```

4. Given the following skeleton of a class:

```
class Rectangle {
  double length;
  double breadth;
  // ...
}
```

Modify the class `Rectangle` so that:

- It is possible to create an object of the class `Rectangle` without specifying the dimensions, and the dimensions in this case should be initialized to the value 1.0.
- It is possible to create a rectangle whose dimensions can be specified.
- The dimensions of a rectangle are never less than 0.0.

The class should offer methods for getting and setting the dimensions of a rectangle. The class should also offer methods to compute the area and the perimeter of a rectangle.

Write a client to test the class `Rectangle`.

5. In a program we will write a class to represent water bottles. The water bottles have the following properties:

- All bottles have a maximum capacity, measured in litres.
- All bottles contain a quantity of water at any given time, also measured in litres.

We can perform the following operations on the bottles:

- Fill a bottle completely from the tap.
- Empty a bottle.
- Pour water from one bottle to another bottle.

Write a class called `Bottle` that implements the properties described above. Let the constructor of the class allow the maximum capacity of a bottle to be specified. To begin with, let all new bottles be empty. Which instance variables should be declared in the class `Bottle`?

Implement the following methods:

a `quantity()`, which returns the amount of water in the current bottle at the moment.

b `remaining()`, which returns the amount of water that can be filled in the current bottle before it is full.

c `fillFully()`, which fills the current bottle completely.

d `empty()`, which empties the current bottle.

e `pour(Bottle b)`, which pours water from the bottle `b` to the current bottle. The amount of water poured into the current bottle is limited either by the capacity of the current bottle or by the amount of water in bottle `b`.

Implement auxiliary methods where necessary.

6. Write a client that uses the `Bottle` class from Exercise 5. The client should be able to do the following:

a Create two `Bottle` objects: one two-litre bottle and one seven-litre bottle.

b Fill the seven-litre bottle completely.

c Pour water from the seven-litre bottle to the two-litre bottle.

d Empty the two-litre bottle.

e Print the amount of water in each bottle.

How many litres of water are left in the seven-litre bottle in the end?

7. Based on Exercise 5, write a client that creates two `Bottle` objects: one three-litre bottle and one five-litre bottle.

Find a way of filling the five-litre bottle with only four litres of water, using only combinations of methods `fillFully()`, `empty()` and `pour()` on the two bottles.

8. In Program 7.5 on page 181 the class `EmployeeV4` defines the constants `EmployeeV4.MALE` and `EmployeeV4.FEMALE`. Write a separate source file with the declaration of an enumerated type called `Gender` that defines the enum constants `MALE` and `FEMALE`. Rewrite the classes `EmployeeV4` and `Client4A` to use this enumerated type.