Declarations 3

Chapter Topics

- An overview of declarations that can be specified in a class
- How to declare and initialize variables
- Using default values for instance variables and static variables
- Understanding lifetime of instance variables, static variables, and local variables
- Using arrays: declaration, construction, initialization, and usage of both simple and multidimensional arrays, including anonymous arrays
- Writing methods, usage of the `this` reference in an instance method, and method overloading
- Understanding the role of constructors, usage of the default constructor, and constructor overloading
- Understanding parameter passing, both primitive values and object references, including arrays and array elements; and declaring `final` parameters0
- Declaring and calling methods with variable arity
- Declaring the `main()` method whose execution starts the application
- Passing program arguments to the `main()` method on the command line
- Declaring and using local variable type inference with `var`

Java SE 17 Developer Exam Objectives	
[3.2] Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers ❍ *Declaring normal classes and defining class members and constructors are covered in this chapter.* ❍ *For record classes, see §5.14, p. 299.* ❍ *For instance and static initializers, see Chapter 10, p. 531.*	§3.1, p. 99 *to* §3.8, p. 112
[3.3] Implement overloading, including var-arg methods ❍ *Method overloading and varargs methods are covered in this chapter.* ❍ *For comparing overloading and overriding, see §5.1, p. 202.*	§3.6, p. 108 §3.11, p. 136

[3.4] Understand variable scopes, use local variable type inference, apply encapsulation, and make objects immutable ❍ *Local variable type inference is covered in this chapter.* ❍ *For detailed coverage of lambda expressions, see* **§13.2**, **p. 679**. ❍ *For variable scope, encapsulation, and immutability, see* **Chapter 6**, **p. 323**.	§3.13, p. 142
[5.1] Create Java arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements ❍ *Arrays are covered in this chapter.* ❍ *For Collections and Maps, see* **Chapter 15**, **p. 781**.	§3.9, p. 117
Java SE 11 Developer Exam Objectives	
[1.3] Use local variable type inference, including as lambda parameters ❍ *Local variable type inference is covered in this chapter.* ❍ *For detailed coverage of lambda expressions, see* **§13.2**, **p. 679**.	§3.13, p. 142
[3.2] Define and use fields and methods, including instance, static and overloaded methods	§3.1, p. 99 *to* §3.8, p. 112
[3.3] Initialize objects and their members using instance and static initialiser statements and constructors	§3.4, p. 102 §3.7, p. 109
[5.2] Use a Java array and List, Set, Map and Deque collections, including convenience methods ❍ *Arrays are covered in this chapter.* ❍ *For collections and maps, see* **Chapter 15**, **p. 781**.	§3.9, p. 117

This chapter covers declaration of classes, methods, constructors, and variables, including array types. Other declarations (e.g., modules, packages, interfaces, enum types, and nested types) will be covered in due course in later chapters.

A *declaration* is a language construct that includes an identifier which can be used to refer to this declaration in the program. Examples include a class declaration that can be referred to by its class name and a method declaration that can be referred to by its method name.

3.1 Class Declarations

A class declaration introduces a new *reference type* and has the following syntax:

Click here to view code image

```
class_modifiers class class_name extends_clause implements_clause // Class header
{ // Class body
          field_declarations
          method_declarations
          constructor_declarations
          member_type_declarations
}
```

In the class header, the name of the class is preceded by the keyword `class`. In addition, the class header can specify the following information:

- The following *class modifiers*:
  ❍ *Access modifiers*: `public`, `protected`, `private` (§6.5, p. 345)
  ❍ *Non-access class modifiers*: `abstract` (§5.4, p. 218), `final` (§5.5, p. 225), `static` (§9.2, p. 495), `sealed` and `non-sealed` (§5.15, p. 311)
- Any class it *extends* using the `extends` clause (§5.1, p. 191)
- Any interfaces it *implements* using the `implements` clause (§5.6, p. 237)

The class body, enclosed in curly brackets (`{}`), can contain *member declarations*, which comprise the following:

- *Field declarations* (p. 102)
- *Method declarations* (p. 100)
- *Constructor declarations* (p. 109)
- *Member type declarations* (§9.1, p. 491)

The declarations can appear in any order in the class body. The only mandatory parts of the class declaration syntax are the keyword `class`, the class name, and the class body curly brackets (`{}`), as exemplified by the following class declaration:

```
class X { }
```

To understand which code can be legally declared in a class, we distinguish between *static context* and *non-static context*. A static context is defined by static methods, static field initializers, and static initializer blocks. A non-static context is defined by instance methods, instance field initializers, instance initializer blocks, and constructors. By *static code*, we mean expressions and statements in a static context; by *non-static code*, we mean expressions and statements in a non-static context. One crucial difference between the two contexts is that *static code in a class can only refer to other static members* in the class, whereas *non-static code can refer to any member of the class*.

3.2 Method Declarations

A method declaration has the following syntax:

Click here to view code image

```
method_modifiers return_type method_name
              (formal_parameter_list) throws_clause // Method header
{ // Method body
      local_variable_declarations
      statements
}
```

In addition to the name of the method, the method header can specify the following information:

- The following *method modifiers*:
  ❍ *Access modifiers*: `public`, `protected`, `private` (§6.5, p. 347)
  ❍ *Non-access method modifiers*: `static` (p. 115), `abstract` (§5.4, p. 224), `final` (§5.5, p. 226), `synchronized` (§22.4, p. 1388)
- The *type* of the *return value*, or `void` if the method does not return any value
  A non-`void` method must either use a `return` statement (§4.13, p. 184) to return a value or throw an exception to terminate its execution.
- A *formal parameter list*
  The *formal parameter list* is a comma-separated list of parameters for passing information to the method when the method is invoked by a *method call* (p. 127). An empty parameter list must be specified by `( )`. Each parameter is a simple variable declaration consisting of its type and name:
  optional_parameter_modifier type parameter_name
  The parameter names are local to the method (§6.6, p. 354). The *optional parameter modifier* `final` is discussed in §3.10, p. 135. It is recommended to use the `@param` tag in a Javadoc comment to document the formal parameters of a method.

The *type* in the parameter declaration *cannot* be designated by the `var` reserved type name, as illustrated in the method declaration below. At compile time, it is not possible to determine the type of the `newSpeed` formal parameter.

Click here to view code image

```
void setSpeed(var newSpeed) {} // var not permitted. Compile-time error!
```

- Any *exceptions* thrown by the method, which are specified in a `throws` clause (§7.5, p. 388)

The method body is a *block* (`{}`) containing the *local variable declarations* (p. 102) and the *statements* (p. 101) of the method. The `return` statement in the body of a method is of particular importance as it terminates the execution of the method and can optionally return a value to the caller of the method (§4.13, p. 184).

The mandatory parts of a method declaration are the return type, the method name, and the method body curly brackets (`{}`), as exemplified by the following method declaration:

```
void noAction() {}
```

Member methods are characterized as one of two types: *instance methods* (p. 106) and *static methods* (p. 112).

The *signature* of a method comprises the name of the method and the types of the formal parameters only. The following method:

Click here to view code image

```
double cubeVolume(double length, double width, double height) {}
```

has the signature:

Click here to view code image

```
cubeVolume(double, double, double)
```

3.3 Statements

Statements in Java can be grouped into various categories. Variable declarations, optionally specified with an initialization expression, are called *declaration statements* (p. 102). Other basic forms of statements are *control flow statements* (Chapter 4, p. 151) and *expression statements*.

An *expression statement* is an expression terminated by a semicolon (`;`). Any value returned by the expression is discarded. Only certain types of expressions have meaning as statements:

- Assignments (§2.7, p. 54)
- Increment and decrement operators (§2.10, p. 69)
- Method calls (p. 127)
- Object creation expressions with the `new` operator (§2.19, p. 92)

A solitary semicolon denotes the *empty statement*, which does nothing.

A block, `{}`, is a *compound* statement that can be used to group zero or more local declarations and statements (§6.6, p. 354). Blocks can be nested, since a block is a statement that can contain other statements. A block can be used in any context where a simple statement is permitted. The compound statement that is embodied in a block begins at the left curly bracket (`{`) and ends with a matching right curly bracket (`}`). Such a block must not be confused with an array initializer in declaration statements (p. 119).

Labeled statements are discussed in §4.10, p. 179.

3.4 Variable Declarations

A *variable* stores a value of a particular type. A variable has a name, a type, and a value associated with it. In Java, variables can store only values of primitive data types and reference values of objects. Variables that store reference values of objects are called *reference variables* (or *object references* or simply *references*).

We distinguish between two kinds of variables: *field variables* and *local variables*. Field variables are variables that are declared in *type declarations* (*classes*, *interfaces*, and *enums*). Local variables are variables that are declared in *methods*, *constructors*, and *blocks*. Local variables declared with the `var` type name are discussed in §3.13, p. 142.

Declaring and Initializing Variables

Variable declarations (technically called *declaration statements*) are used to *declare* variables, meaning they are used to specify the type and the name of variables. This implicitly determines their memory allocation and the values that can be stored in them. Examples of declaring variables that can store primitive values follow:

Click here to view code image

```
char a, b, c;             // a, b, and c are character variables.
double area;              // area is a floating-point variable.
boolean flag;             // flag is a boolean variable.
```

The first declaration is equivalent to the following three declarations:

```
char a;
char b;
char c;
```

A declaration can also be combined with an *initialization expression* to specify an appropriate initial value for the variable.

Click here to view code image

```
int i = 10,              // i is an int variable with initial value 10.
    j = 0b101;           // j is an int variable with initial value 5.
long big = 2147483648L;   // big is a long variable with specified initial value.
```

Reference Variables

A *reference variable* can store the reference value of an object, and can be used to manipulate the object denoted by the reference value.

A variable declaration that specifies a *reference type* (i.e., a class, an array, an interface name, or an enum type) declares a reference variable. Analogous to the declaration of variables of primitive data types, the simplest form of reference variable declaration specifies the name and the reference type only. The declaration determines which objects can be referenced by a reference variable. Before we can use a reference variable to manipulate an object, it must be declared and initialized with the reference value of the object.

Click here to view code image

```
Pizza yummyPizza;   // Variable yummyPizza can reference objects of class Pizza.
Hamburger bigOne,   // Variable bigOne can reference objects of class Hamburger,
          smallOne; // and so can variable smallOne.
```

It is important to note that the preceding declarations do not create any objects of class `Pizza` or `Hamburger`. Rather, they simply create variables that can store reference values of objects of the specified classes.

A declaration can also be combined with an *initializer expression* to create an object whose reference value can be assigned to the reference variable:

Click here to view code image

```
Pizza yummyPizza = new Pizza("Hot&Spicy"); // Declaration statement
```

The reference variable `yummyPizza` can reference objects of class `Pizza`. The keyword `new`, together with the *constructor call* `Pizza("Hot&Spicy")`, creates an object of the class `Pizza`. The reference value of this object is assigned to the variable `yummyPizza`. The newly created object of class `Pizza` can now be manipulated through the reference variable `yummyPizza`.

Initial Values for Variables

This section discusses what value, if any, is assigned to a variable when no explicit initial value is provided in the declaration.

Default Values for Fields

Default values for fields of primitive data types and reference types are listed in **Table 3.1**. The value assigned depends on the type of the field.

Table 3.1 *Default Values*

Data type	Default value
`boolean`	`false`
`char`	`'\u0000'`
Integer (`byte`, `short`, `int`, `long`)	`0L` for `long`, `0` for others
Floating-point (`float`, `double`)	`0.0F` or `0.0D`
Reference types	`null`

If no explicit initialization is provided for a static variable, it is initialized with the default value of its type when the class is loaded. Similarly, if no initialization is provided for an instance variable, it is initialized with the default value of its type when the class is instantiated. The fields of reference types are always initialized with the null reference value if no initialization is provided.

Example 3.1 illustrates the default initialization of fields. Note that static variables are initialized when the class is loaded the first time, and instance variables are initialized accordingly in *every* object created from the class `Light`.

Example 3.1 *Default Values for Fields*

Click here to view code image

```
public class Light {
  // Static variable
  static int counter;      // Default value 0 when class is loaded

  // Instance variables:
  int     noOfWatts = 100; // Explicitly set to 100
  boolean indicator;       // Implicitly set to default value false
  String  location;        // Implicitly set to default value null

  public static void main(String[] args) {
    Light bulb = new Light();
    System.out.println("Static variable counter:     " + Light.counter);
    System.out.println("Instance variable noOfWatts: " + bulb.noOfWatts);
    System.out.println("Instance variable indicator: " + bulb.indicator);
    System.out.println("Instance variable location:  " + bulb.location);
  }
}
```

Output from the program:

Click here to view code image

```
Static variable counter:     0
Instance variable noOfWatts: 100
Instance variable indicator: false
Instance variable location:  null
```

Initializing Local Variables of Primitive Data Types

Local variables are variables that are declared in *methods*, *constructors*, and *blocks*. They are *not* initialized implicitly when they are allocated memory at method invocation—that is, when the execution of a method begins. The same applies to local variables in constructors and blocks. Local variables must be explicitly initialized before being used. The compiler will report an error only if an attempt is made to *use* an uninitialized local variable.

Example 3.2 *Flagging Uninitialized Local Variables of Primitive Data Types*

Click here to view code image

```
public class TooSmartClass {
  public static void main(String[] args) {
    double weight = 10.0, thePrice;                    // (1) Local variables

    if (weight <  10.0) thePrice = 20.50;
    if (weight >  50.0) thePrice = 399.00;
    if (weight >= 10.0) thePrice = weight * 10.0;      // (2) Always executed
    System.out.println("The price is: " + thePrice);   // (3) Compile-time error!
  }
}
```

In **Example 3.2**, the compiler complains that the local variable `thePrice` used in the print statement at (3) may not have been initialized. If allowed to compile and execute, the local variable `thePrice` will get the value `100.0` in the last `if` statement at (2), before it is used in the print statement. However, in **Example 3.2**, the compiler cannot guarantee that code in the body of any of the `if` statements will be executed and thereby the local variable `thePrice` is initialized, as it cannot determine whether the condition in any of the `if` statements is `true` at compile time.

We will not go into the details of *definite assignment analysis* that the compiler performs to guarantee that a local variable is initialized before it is used. In essence, the compiler determines whether a variable is initialized on a path of control flow from where it is declared to where it is used. This analysis can at times be conservative, as in **Example 3.2**.

The program will compile correctly if the local variable `thePrice` is initialized in the declaration, or if an unconditional assignment is made to it. Replacing the declaration of the local variables at (1) in **Example 3.2** with the following declaration solves the problem:

Click here to view code image

```
double weight = 10.0, thePrice = 0.0; //    (1') Both local variables initialized
```

Replacing the condition in any of the `if` statements with a *constant expression* that evaluates to `true` will also allow the compiler to ensure that the local variable `thePrice` is initialized before use in the print statement.

Initializing Local Reference Variables

Local reference variables are bound by the same initialization rules as local variables of primitive data types.

Example 3.3 *Flagging Uninitialized Local Reference Variables*

Click here to view code image

```
public class VerySmartClass {
  public static void main(String[] args) {
    String importantMessage;       // Local reference variable

    System.out.println("The message length is: " +
                         importantMessage.length());  // Compile-time error!
  }
}
```

In Example 3.3, the compiler complains that the local variable `importantMessage` used in the `println` statement may not be initialized. If the variable `important-Message` is set to the value `null`, the program will compile. However, a runtime error (`NullPointerException`) will occur when the code is executed because the variable `im-portantMessage` will not denote any object. The golden rule is to ensure that a reference variable, whether local or not, is assigned a reference value denoting an object before it is used—that is, to ensure that it does not have the value `null`.

The program compiles and runs if we replace the declaration with the following declaration of the local variable, which creates a string literal and assigns its reference value to the local reference variable `importantMessage`:

Click here to view code image

```
String importantMessage = "Initialize before use!";
```

Arrays and their default values are discussed in §3.9, p. 119.

Lifetime of Variables

The *lifetime* of a variable—that is, the time a variable is accessible during execution—is determined by the context in which it is declared. The lifetime of a variable, which is also called its *scope*, is discussed in more detail in §6.6, p. 352. We distinguish among the lifetime of variables in the following three contexts:

- *Instance variables*: members of a class, which are created for each object of the class. In other words, every object of the class will have its own copies of these variables, which are local to the object. The values of these variables at any given time constitute the *state* of the object. Instance variables exist as long as the object they belong to is in use at runtime.
- *Static variables*: members of a class, but which are not created for any specific object of the class, and therefore, belong only to the class. They are created when the class is loaded at runtime and exist as long as the class is available at runtime.

- *Local variables* (also called *method automatic variables*): declared in methods, constructors, and blocks, and created for each execution of the method, constructor, or block. After the execution of the method, constructor, or block completes, local variables are no longer accessible.

3.5 Instance Methods and the Object Reference `this`

Instance methods belong to every object of the class and can be invoked only on objects. All members defined in the class, both static and non-static, are accessible in the context of an instance method. The reason is that all instance methods are passed an implicit reference to the *current object*—that is, the object on which the method is being invoked.

The current object can be referenced in the body of the instance method by the keyword `this`. In the body of the method, the `this` reference can be used like any other object reference to access members of the object. In fact, the keyword `this` can be used in any non-static context. The `this` reference can be used as a normal reference to reference the current object, but the reference cannot be modified—it is a `final` reference (§5.5, p. 225).

The `this` reference to the current object is useful in situations where a local variable hides, or *shadows*, a field with the same name. In **Example 3.4**, the two parameters `noOfWatts` and `indicator` in the constructor of the `Light` class have the same names as the fields in the class. The example also declares a local variable `location`, which has the same name as one of the fields. The reference `this` can be used to distinguish the fields from the local variables. At (1), the `this` reference is used to identify the field `noOfWatts`, which is assigned the value of the parameter `noOfWatts`. Without the `this` reference at (2), the value of the parameter `indicator` is assigned back to this parameter, and not to the field by the same name, resulting in a logical error. Similarly at (3), without the `this` reference, it is the local variable `location` that is assigned the value of the parameter `site`, and not the field with the same name.

Example 3.4 *Using the* `this` *Reference*

Click here to view code image

```
public class Light {
  // Fields:
  int     noOfWatts;      // Wattage
  boolean indicator;      // On or off
  String  location;       // Placement

  // Constructor:
  public Light(int noOfWatts, boolean indicator, String site) {
    String location;
```

```
    this.noOfWatts = noOfWatts;   // (1) Assignment to field
    indicator = indicator;        // (2) Assignment to parameter
    location = site;              // (3) Assignment to local variable
    this.superfluous();           // (4)
    superfluous();                // equivalent to call at (4)
  }

  // Instance method:
  public void superfluous() {
    System.out.printf("Current object: %s%n", this); // (5)
  }

  // Static method:
  public static void main(String[] args) {
    Light light = new Light(100, true, "loft");
    System.out.println("No. of watts: " + light.noOfWatts);
    System.out.println("Indicator:    " + light.indicator);
    System.out.println("Location:     " + light.location);
  }
}
```

Probable output from the program:

Click here to view code image

```
Current object: Light@1bc4459
Current object: Light@1bc4459
No. of watts: 100
Indicator:    false
Location:     null
```

If a member is not shadowed by a local declaration, the simple name `member` is considered a shorthand notation for `this.member`. In particular, the `this` reference can be used explicitly to invoke other methods in the class. This usage is illustrated at (4) in **Example 3.4**, where the method `superfluous()` is called.

If, for some reason, a method needs to pass the current object to another method, it can do so using the `this` reference. This approach is illustrated at (5) in **Example 3.4**, where the current object is passed to the `printf()` method. The `printf()` method prints the text representation of the current object (which comprises the name of the class of the current object and the hexadecimal representation of the current object's hash code). The *hash code* of an object is an `int` value that uniquely identifies the object.

Note that the `this` reference cannot be used in a static context, as static code is not executed in the context of any object.

3.6 Method Overloading

Each method has a *signature*, which comprises the name of the method plus the types and order of the parameters in the formal parameter list. Several method implementations may have the same name, as long as the method signatures differ. This practice is called *method overloading*. Because overloaded methods have the same name, their parameter lists must be different.

Rather than inventing new method names, method overloading can be used when the same logical operation requires multiple implementations. The Java SE Platform API makes heavy use of method overloading. For example, the class `java.lang.Math` contains an overloaded method `min()`, which returns the minimum of two numeric values.

Click here to view code image

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

In the following examples, five implementations of the method `methodA` are shown:

Click here to view code image

```
void methodA(int a, double b) { /* ... */ }      // (1)
int  methodA(int a)           { return a; }      // (2)
int  methodA()                { return 1; }      // (3)
long methodA(double a, int b) { return b; }      // (4)
long methodA(int x, double y) { return x; }      // (5) Not OK.
```

The corresponding signatures of the five methods are as follows:

Click here to view code image

```
methodA(int, double)      1'
methodA(int)j             2': Number of parameters
methodA()                 3': Number of parameters
methodA(double, int)      4': Order of parameters
methodA(int, double)      5': Same as 1'
```

The first four implementations of the method named `methodA` are overloaded correctly, each time with a different parameter list and, therefore, different signatures. The declaration at (5) has the same signature `methodA(int, double)` as the declaration at (1), and therefore, is not a valid overloading of this method.

[Click here to view code image](#)

```
void bake(Cake k)  { /* ... */ }                    // (1)
void bake(Pizza p) { /* ... */ }                    // (2)

int     halfIt(int a) { return a/2; }               // (3)
double  halfIt(int a) { return a/2.0; }             // (4) Not OK. Same signature.
```

The method named `bake` is correctly overloaded at (1) and (2), with two different parameter lists. In the implementation, changing just the return type (as shown at (3) and (4) in the preceding example) is not enough to overload a method and will be flagged as a compile-time error. The parameter list in the declarations must be different.

Only methods declared in the same class and those that are inherited by the class can be overloaded. Overloaded methods should be considered to be individual methods that just happen to have the same name. Methods with the same name are allowed, since methods are identified by their signature. At compile time, the right implementation of an overloaded method is chosen, based on the signature of the method call. Details of method overloading resolution can be found in §5.10, p. 265. Method overloading should not be confused with *method overriding* (§5.1, p. 196).

3.7 Constructors

The main purpose of constructors is to set the initial state of an object, when the object is created by using the `new` operator.

The following simplified syntax is the canonical declaration of a constructor:

[Click here to view code image](#)

```
access_modifier class_name (formal_parameter_list)
                                  throws_clause   // Constructor header
{ // Constructor body
     local_variable_declarations
     statements
}
```

Constructor declarations are very much like method declarations. However, the following restrictions on constructors should be noted:

- Modifiers other than an access modifier are not permitted in the constructor header. For more on access modifiers for constructors, see §6.5, p. 345.
- Constructors cannot return a value, and therefore, do not specify a return type, not even `void`, in the constructor header. But their declaration can use the `return` state-

ment (without the return value) in the constructor body (§4.13, p. 184).

- The constructor name must be the same as the class name.

Class names and method names exist in different *namespaces*. Thus there are no name conflicts in Example 3.5, where a method declared at (2) has the same name as the constructor declared at (1). The method `Name()` at (2) is also breaking the convention of starting a method name with a lowercase character. A method must always specify a return type, whereas a constructor does not. However, using such naming schemes is strongly discouraged.

A constructor that has no parameters, like the one at (1) in Example 3.5, is called a *no-argument constructor*.

Example 3.5 *Namespaces*

Click here to view code image

```
public class Name {

  Name() {                          // (1) No-argument constructor
    System.out.println("Constructor");
  }

  void Name() {                     // (2) Instance method
    System.out.println("Method");
  }

  public static void main(String[] args) {
    new Name().Name();              // (3) Constructor call followed by method call
  }
}
```

Output from the program:

```
Constructor
Method
```

The Default Constructor

If a class does not specify *any* constructors, then a *default constructor* is generated for the class by the compiler. The default constructor is equivalent to the following implementation:

Click here to view code image

```
class_name() { super(); }   // No parameters. Calls superclass constructor.
```

A default constructor is a no-argument constructor. The only action taken by the default constructor is to call the superclass constructor. This ensures that the inherited state of the object is initialized properly (§5.3, p. 209). In addition, all instance variables in the object are set to the default value of their type, barring those that are initialized by an initialization expression in their declaration.

In the following code, the class `Light` does not specify any constructors:

Click here to view code image

```
class Light {
  // Fields:
  int     noOfWatts;       // Wattage
  boolean indicator;       // On or off
  String  location;        // Placement

  // No constructors
  //...
}

class Greenhouse {
  // ...
  Light oneLight = new Light();     // (1) Call to default constructor
}
```

In this code, the following default constructor is called when a `Light` object is created by the object creation expression at (1):

```
Light() { super(); }
```

Creating an object using the `new` operator with the default constructor, as at (1), will initialize the fields of the object to their *default values* (i.e., the fields `noOfWatts`, `indicator`, and `location` in a `Light` object will be initialized to `0`, `false`, and `null`, respectively).

A class can choose to provide its own constructors, rather than relying on the default constructor. In the following example, the class `Light` provides a no-argument constructor at (1).

Click here to view code image

```
class Light {
  // ...
  Light() {                        // (1) No-argument constructor
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }
  //...
}
class Greenhouse {
  // ...
  Light extraLight = new Light();   // (2) Call to no-argument constructor
}
```

The no-argument constructor ensures that any object created with the object creation expression `new Light()`, as at (2), will have its fields `noOfWatts`, `indicator`, and `location` initialized to `50`, `true`, and `"X"`, respectively.

If a class defines *any* constructor, the default constructor is not generated. If such a class requires a no-argument constructor, it must provide its own implementation, as in the preceding example. In the next example, the class `Light` does not provide a no-argument constructor, but rather includes a non-zero argument constructor at (1). It is called at (2) when an object of the class `Light` is created with the `new` operator. Any attempt to call the default constructor will be flagged as a compile-time error, as shown at (3).

Click here to view code image

```
class Light {
  // ...
  // Only non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {          // (1)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}

class Greenhouse {
  // ...
  Light moreLight  = new Light(100, true, "Greenhouse");// (2) OK
  Light firstLight = new Light();                       // (3) Compile-time error
}
```

Overloaded Constructors

Like methods, constructors can be overloaded. Since the constructors in a class all have the same name as the class, their signatures are differentiated by their parameter lists. In the following example, the class `Light` now provides explicit implementation of the no-argument constructor at (1) and that of a non-zero argument constructor at (2). The constructors are overloaded, as is evident by their signatures. The non-zero argument constructor at (2) is called when an object of the class `Light` is created at (4), and the no-argument constructor is likewise called at (3). Overloading of constructors allows appropriate initialization of objects on creation, depending on the constructor invoked (see chaining of constructors in §5.3, p. 209). It is recommended to use the `@param` tag in a Javadoc comment to document the formal parameters of a constructor.

Click here to view code image

```
class Light {
  // ...
  // No-argument constructor:
  Light() {                                                   // (1)
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) { // (2)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}

class Greenhouse {
  // ...
  Light firstLight = new Light();                         // (3) OK. Calls (1)
  Light moreLight  = new Light(100, true, "Greenhouse"); // (4) OK. Calls (2)
}
```

3.8 Static Member Declarations

In this section we look at static members in classes, but in general, the keyword `static` is used in the following contexts:

- Declaring `static` fields in classes (p. 113), enum types (§5.13, p. 290) and interfaces (§5.6, p. 254)

- Declaring `static` methods in classes (p. 115), enum types (§5.13, p. 290) and interfaces (§5.6, p. 251)
- Declaring `static` initializer blocks in classes (§10.7, p. 545) and enum types (§5.13, p. 290)
- Declaring nested `static` member types (§9.2, p. 495)

Static Members in Classes

Static members belong to the class in which they are declared and are not part of any instance of the class. The declaration of `static` members is prefixed by the keyword `static` to distinguish them from instance members.

Static code inside a class can access a `static` member in the following three ways:

- By the `static` member's simple name
- By using the class name with the `static` member's name
- By using an object reference of the `static` member's class with the `static` member's name

Depending on the access modifier of the `static` members declared in a class, clients can only access these members by using the class name or using an object reference of their class.

The class need not be instantiated to access its `static` members. This is in contrast to instance members of the class which can only be accessed by references that actually refer to an instance of the class.

Static Fields in Classes

Static fields (also called *static variables* and *class variables*) exist only in the class in which they are defined. When the class is loaded, `static` fields are initialized to their default values if no explicit initialization is specified. They are not created when an instance of the class is created. In other words, the values of these fields are not a part of the state of any object. Static fields are akin to global variables that can be shared with all objects of the class and with other clients, if necessary.

Example 3.6 *Accessing Static Members in a Class*

Click here to view code image

```
// File: StaticTest.java
import static java.lang.System.out;

class Light {
```

```
  // Static field:
  static int counter;                  // (1) No initializer expression

  // Static method:
  public static void printStatic() {
    Light myLight = null;
    out.printf("%s, %s, %s%n", counter, Light.counter, myLight.counter); // (2)
    long counter = 10;                 // (3) Local variable shadows static field
    out.println("Local counter: " + counter);       // (4) Local variable accessed
    out.println("Static counter: " + Light.counter);// (5) Static field accessed

//  out.println(this.counter);         // (6) Cannot use this in static context
//  printNonStatic();                  // (7) Cannot call non-static method
  }

  // Non-static method:
  public void printNonStatic() {
   out.printf("%s, %s, %s%n", counter, this.counter, Light.counter);     // (8)
  }
}
//______________________________________________________________________________
public class StaticTest {              // Client of class Light
  public static void main(String[] args) {
    Light.counter++;                   // (9) Using class name
    Light dimLight = null;
    dimLight.counter++;                // (10) Using object reference

    out.print("Light.counter == dimLight.counter: ");
    out.println(Light.counter == dimLight.counter);//(11) Aliases for static field

    out.println("Calling static method using class name:");
    Light.printStatic();               // (12) Using class name
    out.println("Calling static method using object reference:");
    dimLight.printStatic();            // (13) Using object reference
  }
}
```

Output from the program:

Click here to view code image

```
Light.counter == dimLight.counter: true
Calling static method using class name:
2, 2, 2
Local counter: 10
Static counter: 2
Calling static method using object reference:
2, 2, 2
```

```
Local counter: 10
Static counter: 2
```

In **Example 3.6**, the `static` field `counter` at (1) will be initialized to the default value `0` when the class is loaded at runtime, since no initializer expression is specified. The print statement at (2) in the `static` method `printCount()` shows how this `static` field can be accessed in three different ways, respectively: simple name `counter`, the class name `Light`, and object reference `myLight` of class `Light,` although no object has been created.

Shadowing of fields by local variables is different from *hiding* of fields by field declarations in subclasses. In **Example 3.6**, a local variable is declared at (3) that has the same name as the `static` field. Since this local variable *shadows* the `static` field, the simple name at (4) now refers to the local variable, as shown by the output from the program. The shadowed `static` field can of course be accessed using the class name, as shown at (5). It is the local variable that is accessed by its simple name as long as it is in scope.

Trying to access the `static` field with the `this` reference at (6) results in a compile-time error, since the `this` reference cannot be used in static code. Invoking the non-`static` method at (7) also results in a compile-time error, since static code cannot refer to non-`static` members by its simple name in the class.

The print statement at (8) in the method `printNonStatic()` illustrates referring to `static` members in non-static code: It refers to the `static` field counter by its simple name, with the `this` reference, and using the class name.

In **Example 3.6**, the class `StaticTest` is a client of the class `Light`. The client must use the class name or an object reference of class `Light` at (9) and (10), respectively, to access the `static` field `counter` in the class `Light`. The result from the print statement at (11) shows that these two ways of accessing a `static` field are equivalent.

Static Methods in Classes

Static methods are also known as *class methods*. A `static` method in a class can directly access other `static` members in the class by their simple name. It cannot access instance (i.e., non-`static`) members of the class directly, as there is no notion of an object associated with a `static` method.

A typical `static` method might perform some task on behalf of the whole class or for objects of the class. Static methods are often used to implement *utility classes* that provide common and frequently used functions. A good example of a utility class is the `java.lang.Math` class in the Java platform SE API that provides common mathematical functions.

Static methods can be overloaded analogous to instance methods. Static methods in a superclass cannot be overridden in a subclass as instance methods can, but they can be *hidden* by `static` methods in a subclass (§5.1, p. 203).

A type parameter of a generic class or interface cannot be used in a `static` method (§11.2, p. 567).

Example 3.6 shows how the `static` method `printCount()` of the class `Light` can be invoked using the class name and via an object reference of the class `Light` at (12) and (13), respectively.

Review Questions

3.1 In which of these variable declarations will the variable remain uninitialized unless it is explicitly initialized?

Select the one correct answer.

a. Declaration of an instance variable of type `int`

b. Declaration of a static variable of type `float`

c. Declaration of a local variable of type `float`

d. Declaration of a static variable of type `Object`

e. Declaration of an instance variable of type `int[]`

3.2 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Init {

  String title;
  boolean published;

  static int total;
  static double maxPrice;

  public static void main(String[] args) {
    Init initMe = new Init();
    double price;
    if (true)
      price = 100.00;
```

```
    System.out.println("|" + initMe.title + "|" + initMe.published + "|" +
                       Init.total + "|" + Init.maxPrice + "|" + price + "|");
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will print `|null|false|0|0.0|0.0|` at runtime.

c. The program will print `|null|true|0|0.0|100.0|` at runtime.

d. The program will print `| |false|0|0.0|0.0|` at runtime.

e. The program will print `|null|false|0|0.0|100.0|` at runtime.

3.3 Given that the following pairs of method are declared in the same class, which of the following statements are true?

Click here to view code image

```
void fly(int distance) {}
int  fly(int time, int speed) { return time*speed; }

void fall(int time) {}
int  fall(int distance) { return distance; }

void glide(int time) {}
void Glide(int time) {}
```

Select the two correct answers.

a. The first pair of methods will compile and will overload the method name `fly`.

b. The second pair of methods will compile and will overload the method name `fall`.

c. The third pair of methods will compile and will overload the method name `glide`.

d. The first pair of methods will fail to compile.

e. The second pair of methods will fail to compile.

f. The third pair of methods will fail to compile.

3.4 Which of the following statements are true? Select the two correct answers.

a. A class must define a constructor.

b. A constructor can be declared `private`.

c. A constructor can return a value.

d. A constructor must initialize all fields when a class is instantiated.

e. A constructor can access the non-static members of a class.

3.5 What will be the result of compiling the following program?

Click here to view code image

```
public class MyClass {
  long var;

  public void MyClass(long param) { var = param; }  // (1)

  public static void main(String[] args) {
    MyClass a, b;
    a = new MyClass();                              // (2)
    b = new MyClass(5);                             // (3)
  }
}
```

Select the one correct answer.

a. A compile-time error will occur at (1).

b. A compile-time error will occur at (2).

c. A compile-time error will occur at (3).

d. The program will compile without errors.

3.6 Which statement is true?

Select the one correct answer.

a. A `static` method can call other non-`static` methods in the same class by using the `this` keyword.

b. A class may contain both `static` and non-`static` variables, and both `static` and non-`static` methods.

c. Each object of a class has its own instance of the `static` variables declared in the class.

d. Instance methods may access local variables of `static` methods.

e. All methods in a class are implicitly passed the `this` reference as an argument, when invoked.

3.9 Arrays

An *array* is a data structure that defines an indexed collection with a fixed number of data elements that all have the *same type*. A position in the array is indicated by a non-negative integer value called the *index*. An element at a given position in the array is accessed using the index. The size of an array is fixed and cannot be changed after the array has been created.

In Java, arrays are objects. Arrays can be of primitive data types or reference types. In the former case, all elements in the array are of a specific primitive data type. In the latter case, all elements are references of a specific reference type. References in the array can then denote objects of this reference type or its subtypes. Each array object has a `public final` field called `length`, which specifies the array size (i.e., the number of elements the array can accommodate). The first element is always at index 0 and the last element at index $n - 1$, where n is the value of the `length` field in the array.

Simple arrays are *one-dimensional arrays*—that is, a simple list of values. Since arrays can store reference values, the objects referenced can also be array objects. Thus a multi-dimensional arrays is implemented as an *array of arrays* (p. 124).

Passing array references as parameters is discussed in §3.10, p. 127. Type conversions for array references on assignment and on method invocation are discussed in §5.9, p. 261, and §5.10, p. 265, respectively.

Declaring Array Variables

A one-dimensional array variable declaration has either of the following syntaxes:

```
element_type[] array_name;
```

or

```
element_type array_name[];
```

where *element_type* can be a primitive data type or a reference type. The array variable *array_name* has the type *element_type* `[]`. Note that the array size is not specified. As a consequence, the array variable *array_name* can be assigned the reference value of an array of any length, as long as its elements have *element_type*.

It is important to understand that the declaration does not actually create an array. Instead, it simply declares a *reference* that can refer to an array object. The `[]` notation can also be specified after a variable name to declare it as an array variable, but then it applies to just that variable.

Click here to view code image

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

These two declarations declare `anIntArray` and `mediumPizzas` to be reference variables that can refer to arrays of `int` values and arrays of `Pizza` objects, respectively. The variable `largePizzas` can denote an array of `Pizza` objects, but the variable `oneInteger` cannot denote an array of `int` values—it is a simple variable of the type `int`.

An array variable that is declared as a field in a class, but is not explicitly initialized to any array, will be initialized to the default reference value `null`. This default initialization does *not* apply to *local* reference variables, and therefore, does not apply to local array variables either. This behavior should not be confused with initialization of the elements of an array during array construction.

Constructing an Array

An array can be constructed for a fixed number of elements of a specific type, using the `new` operator. The reference value of the resulting array can be assigned to an array variable of the corresponding type. The syntax of the *array creation expression* is shown on the right-hand side of the following assignment statement:

Click here to view code image

```
array_name = new element_type[array_size];
```

The minimum value of *array_size* is `0`; in other words, zero-length arrays can be constructed in Java. If the array size is negative, a `NegativeArraySizeException` is thrown at runtime.

Given the declarations

Click here to view code image

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

the three arrays in the declarations can be constructed as follows:

Click here to view code image

```
anIntArray   = new int[10];          // array for 10 integers
mediumPizzas = new Pizza[5];         // array of 5 pizzas
largePizzas  = new Pizza[3];         // array of 3 pizzas
```

The array declaration and construction can be combined.

Click here to view code image

```
element_type1[] array_name = new element_type2[array_size];
```

In the preceding syntax, the array type *element_type*$_2$ `[]` must be *assignable* to the array type *element_type*$_1$ `[]` (§5.8, p. 261). When the array is constructed, all of its elements are initialized to the default value for *element_type*$_2$. This is true for both member and local arrays when they are constructed.

In the following examples, the code constructs the array, and the array elements are implicitly initialized to their default values. For example, all elements of the array `anIntArray` get the value `0`, and all elements of the array `mediumPizzas` get the value `null` when the arrays are constructed.

Click here to view code image

```
int[] anIntArray = new int[10];                  // Default element value: 0
Pizza[] mediumPizzas = new Pizza[5];             // Default element value: null
```

The value of the field `length` in each array is set to the number of elements specified during the construction of the array; for example, `mediumPizzas.length` has the value `5`.

Once an array has been constructed, its elements can also be explicitly initialized individually—for example, in a loop. The examples in the rest of this section make use of a loop to iterate over the elements of an array for various purposes.

Initializing an Array

Java provides the means to declare, construct, and explicitly initialize an array in one declaration statement:

Click here to view code image

```
element_type[] array_name = { array_initialize_list };
```

This form of initialization applies to fields as well as to local arrays. The *array_initialize_list* is a comma-separated list of zero or more expressions. Such an array initializer results in the construction and initialization of the array.

Click here to view code image

```
int[] anIntArray = {13, 49, 267, 15, 215};
```

In the declaration statement above, the variable `anIntArray` is declared as a reference to an array of `int` s. The array initializer results in the construction of an array to hold five elements (equal to the length of the list of expressions in the block), where the first element is initialized to the value of the first expression (`13)` , the second element to the value of the second expression (`49)` , and so on.

Click here to view code image

```
Pizza[] pizzaOrder = { new Pizza(), new Pizza(), null };
```

In this declaration statement, the variable `pizzaOrder` is declared as a reference to an array of `Pizza` objects. The array initializer constructs an array to hold three elements. The initialization code sets the first two elements of the array to refer to two `Pizza` objects, while the last element is initialized to the null reference. The reference value of the array of `Pizza` objects is assigned to the reference `pizzaOrder` . Note also that this declaration statement actually creates *three* objects: the array object with three references and the two `Pizza` objects.

The expressions in the *array_initialize_list* are evaluated from left to right, and the array name obviously cannot occur in any of the expressions in the list. In the preceding examples, the *array_initialize_list* is terminated by the right curly bracket, `}` , of the block. The list can also be legally terminated by a comma. The following array has length 2, and not 3:

Click here to view code image

```
Topping[] pizzaToppings = { new Topping("cheese"), new Topping("tomato"), };
```

The declaration statement at (1) in the following code defines an array of four `String` objects, while the declaration statement at (2) shows that a `String` object is not the same as an array of `char`.

Click here to view code image

```
// Array with 4 String objects:
String[] pets = {"crocodiles", "elephants", "crocophants", "elediles"}; // (1)

// Array of 3 characters:
char[] charArray = {'a', 'h', 'a'};    // (2) Not the same as "aha"
```

Using an Array

The array object is referenced by the array name, but individual array elements are accessed by specifying an index with the `[]` operator. The array element access expression has the following syntax:

Click here to view code image

```
array_name [index_expression]
```

Each individual element is treated as a simple variable of the element type. The *index* is specified by the *index_expression*, whose value should be promotable to an `int` value; otherwise, a compile-time error is flagged. Since the lower bound of an array index is always `0`, the upper bound is 1 less than the array size—that is, *array_name* `.length-1`. The `i` th element in the array has index `(i-1)`. At runtime, the index value is automatically checked to ensure that it is within the array index bounds. If the index value is less than `0`, or greater than or equal to *array_name*. `length`, an `ArrayIndexOutOfBoundsException` is thrown. A program can either explicitly check that the index value is within the array index bounds or catch the runtime exception that is thrown if it is invalid (§7.3, p. 375), but an illegal index is typically an indication of a programming error.

In the array element access expression, the *array_name* can be any expression that returns a reference to an array. For example, the expression on the right-hand side of the following assignment statement returns the character `'H'` at index 1 in the character array returned by a call to the `toCharArray()` method of the `String` class:

Click here to view code image

```
char letter = "AHA".toCharArray()[1];     // 'H'
```

The array operator `[]` is used to declare array types (`Topping[]`), specify the array size (`new Topping[3]`), and access array elements (`toppings[1]`). This operator is not used when the array reference is manipulated, such as in an array reference assignment, or when the array reference is passed as an actual parameter in a method call (p. 132).

Example 3.7 shows traversal of arrays using `for` loops (§4.7, p. 174 and p. 176). A `for(;;)` loop at (3) in the `main()` method initializes the local array `trialArray` declared at (2) five times with pseudorandom numbers (from `0.0` to `100.0`), by calling the method `randomize()` declared at (5). The minimum value in the array is found by calling the method `findMinimum()` declared at (6), and is stored in the array `storeMinimum` declared at (1). Both of these methods also use a `for(;;)` loop. The loop variable is initialized to a start value—`0` at (3) and (5), and `1` at (6). The loop condition tests whether the loop variable is less than the length of the array; this guarantees that the loop will terminate when the last element has been accessed. The loop variable is incremented after each iteration to access the next element.

A `for(:)` loop at (4) in the `main()` method is used to print the minimum values from the trials, as elements are read consecutively from the array, without keeping track of an index value.

Example 3.7 *Using Arrays*

Click here to view code image

```
public class Trials {
  public static void main(String[] args) {
    // Declare and construct the local arrays:
    double[] storeMinimum = new double[5];              // (1)
    double[] trialArray = new double[15];               // (2)
    for (int i = 0; i < storeMinimum.length; ++i) {     // (3)
      // Initialize the array.
      randomize(trialArray);

      // Find and store the minimum value.
      storeMinimum[i] = findMinimum(trialArray);
    }

    // Print the minimum values:                           (4)
    for (double minValue : storeMinimum)
      System.out.printf("%.4f%n", minValue);
  }

  public static void randomize(double[] valArray) {     // (5)
```

```
    for (int i = 0; i < valArray.length; ++i)
      valArray[i] = Math.random() * 100.0;
  }

  public static double findMinimum(double[] valArray) {  // (6)
    // Assume the array has at least one element.
    double minValue = valArray[0];
    for (int i = 1; i < valArray.length; ++i)
      minValue = Math.min(minValue, valArray[i]);
    return minValue;
  }
}
```

Probable output from the program:

```
6.9330
2.7819
6.7427
18.0849
26.2462
```

Anonymous Arrays

As shown earlier in this section, the following declaration statement can be used to construct arrays using an array creation expression:

Click here to view code image

```
element_type1[] array_name = new element_type2[array_size];   // (1)

int[] intArray = new int[5];
```

The size of the array is specified in the array creation expression, which creates the array and initializes the array elements to their default values. By comparison, the following declaration statement both creates the array and initializes the array elements to specific values given in the array initializer:

Click here to view code image

```
element_type[] array_name = { array_initialize_list }; // (2)

int[] intArray = {3, 5, 2, 8, 6};
```

However, the array initializer is *not* an expression. Java has another array creation expression, called an *anonymous array*, which allows the concept of the array creation expression from (1) to be combined with the array initializer from (2), so as to create and initialize an array:

Click here to view code image

```
new element_type[] { array_initialize_list }

new int[] {3, 5, 2, 8, 6}
```

This construct has enough information to create a nameless array of a specific type and specific length. Neither the name of the array nor the size of the array is specified. The construct returns the reference value of the newly created array, which can be assigned to references and passed as arguments in method calls. In particular, the following declaration statements are equivalent:

Click here to view code image

```
int[] intArray = {3, 5, 2, 8, 6};                          // (1)
int[] intArray = new int[] {3, 5, 2, 8, 6};                // (2)
```

At (1), an array initializer is used to create and initialize the elements. At (2), an anonymous array expression is used. It is tempting to use the array initializer as an expression—for example, in an assignment statement, as a shortcut for assigning values to array elements in one go. However, this is not allowed; instead, an anonymous array expression should be used. The concept of the anonymous array combines the definition and the creation of the array into one operation.

Click here to view code image

```
int[] daysInMonth;
daysInMonth = {31, 28, 31, 30, 31, 30,
               31, 31, 30, 31, 30, 31};                    // Compile-time error
daysInMonth = new int[] {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // OK
```

In **Example 3.8**, an anonymous array is constructed at (1), and passed as an actual parameter to the static method `findMinimum()` defined at (2). Note that no array name or array size is specified for the anonymous array.

Example 3.8 *Using Anonymous Arrays*

Click here to view code image

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

```
    System.out.println("Minimum value: " +
        findMinimum(new int[] {3, 5, 2, 8, 6}));                     // (1)
  }

  public static int findMinimum(int[] dataSeq) {                     // (2)
    // Assume the array has at least one element.
    int min = dataSeq[0];
    for (int index = 1; index < dataSeq.length; ++index)
      if (dataSeq[index] < min)
        min = dataSeq[index];
    return min;
  }
}
```

Output from the program:

```
Minimum value: 2
```

Multidimensional Arrays

Since an array element can be an object reference and arrays are objects, array elements can themselves refer to other arrays. In Java, an array of arrays can be defined as follows:

Click here to view code image

```
element_type[][]...[] array_name;
```

or

Click here to view code image

```
element_type array_name[][]...[];
```

In fact, the sequence of square bracket pairs, `[]`, indicating the number of dimensions, can be distributed as a postfix to both the element type and the array name. Arrays of arrays are often called *multidimensional arrays*.

The following declarations are all equivalent:

Click here to view code image

```
int[][] mXnArray;      // two-dimensional array
int[]   mXnArray[];    // two-dimensional array
```

```
int     mXnArray[][];  // two-dimensional array
```

It is customary to combine the declaration with the construction of the multidimensional array.

Click here to view code image

```
int[][] mXnArray = new int[4][5];    // 4 x 5 matrix of ints
```

The previous declaration constructs an array `mXnArray` of four elements, where each element is an array (row) of five `int` values. The concept of rows and columns is often used to describe the dimensions of a two-dimensional array, which is often called a *matrix*. However, such an interpretation is not dictated by the Java language.

Each row in the previous matrix is denoted by `mXnArray[i]`, where `0` ≤ `i` < `4.` Each element in the `i` th row, `mXnArray[i]`, is accessed by `mXnArray[i][j]`, where `0` ≤ `j` < `5.` The number of rows is given by `mXnArray.length`, in this case `4`, and the number of values in the `i` th row is given by `mXnArray[i].length`, in this case `5` for all the rows, where `0` ≤ `i` < `4`.

Multidimensional arrays can also be constructed and explicitly initialized using the array initializers discussed for simple arrays. Note that each row is an array that uses an array initializer to specify its values:

Click here to view code image

```
double[][] identityMatrix = {
  {1.0, 0.0, 0.0, 0.0 }, // 1. row
  {0.0, 1.0, 0.0, 0.0 }, // 2. row
  {0.0, 0.0, 1.0, 0.0 }, // 3. row
  {0.0, 0.0, 0.0, 1.0 }  // 4. row
}; // 4 x 4 floating-point matrix
```

Arrays in a multidimensional array need not have the same length; in which case, they are called *ragged arrays*. The array of arrays `pizzaGalore` in the following code has five rows; the first four rows have different lengths but the fifth row is left unconstructed:

Click here to view code image

```
Pizza[][] pizzaGalore = {
  { new Pizza(), null, new Pizza() },    // 1. row is an array of 3 elements.
  { null, new Pizza()},                  // 2. row is an array of 2 elements.
  new Pizza[1],                          // 3. row is an array of 1 element.
  {},                                    // 4. row is an array of 0 elements.
```

```
  null                                // 5. row is not constructed.
};
```

When constructing multidimensional arrays with the `new` operator, the length of the deeply nested arrays may be omitted. In such a case, these arrays are left unconstructed. For example, an array of arrays to represent a room (defined by class `HotelRoom`) on a floor in a hotel on a street in a city can have the type `HotelRoom[][][][]`. From left to right, the square brackets represent indices for street, hotel, floor, and room, respectively. This four-dimensional array of arrays can be constructed piecemeal, starting with the leftmost dimension and proceeding to the rightmost successively.

Click here to view code image

```
HotelRoom[][][][] rooms = new HotelRoom[10][5][][];  // Just streets and hotels.
```

The preceding declaration constructs the array of arrays `rooms` partially with 10 streets, where each street has five hotels. Floors and rooms can be added to a particular hotel on a particular street:

Click here to view code image

```
rooms[0][0]       = new HotelRoom[3][]; // 3 floors in 1st hotel on 1st street.
rooms[0][0][0]    = new HotelRoom[8];   // 8 rooms on 1st floor in this hotel.
rooms[0][0][0][0] = new HotelRoom();    // Initializes 1st room on this floor.
```

The next code snippet constructs an array of arrays `matrix`, where the first row has one element, the second row has two elements, and the third row has three elements. Note that the outer array is constructed first. The second dimension is constructed in a loop that constructs the array in each row. The elements in the multidimensional array will be implicitly initialized to the default `double` value (`0.0D`). In **Figure 3.1**, the array of arrays `matrix` is depicted after the elements have been explicitly initialized.

Click here to view code image

```
double[][] matrix = new double[3][];      // (1) Number of rows.

for (int i = 0; i < matrix.length; ++i)
  matrix[i] = new double[i + 1];          // Construct a row.
```

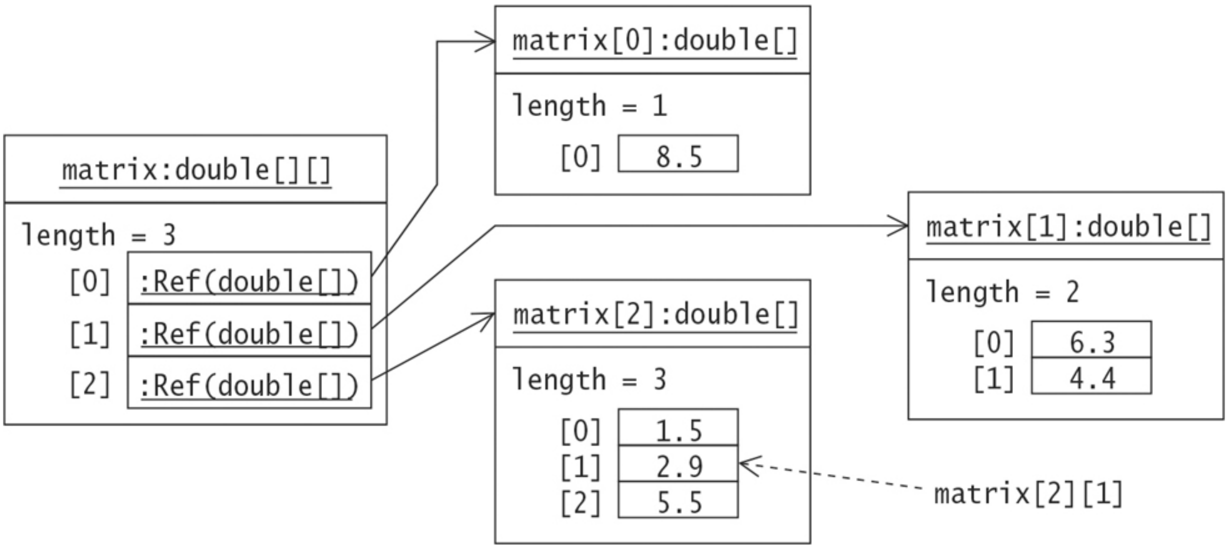

Figure 3.1 *Array of Arrays*

The type of the variable `matrix` is `double[][]` at (1), a two-dimensional array of `double` values. The type of the variable `matrix[i]` (where `0` ≤ `i` < `matrix.length`) is `double[]`, a one-dimensional array of `double` values. The type of the variable `matrix[i][j]` (where `0` ≤ `i` < `matrix.length` and `0` ≤ `j` < `matrix[i].length`) is `double`, a simple variable of type `double`.

Two other ways of initializing such an array of arrays are shown next. The first approach uses array initializers, and the second uses an anonymous array of arrays.

Click here to view code image

```
double[][] matrix2 = {     // (2) Using array initializers.
  {1.0},                   // 1. row
  {1.0, 2.0},              // 2. row
  {1.0, 2.0, 3.0}          // 3. row
};

double[][] matrix3 = new double[][] { // (3) Using an anonymous array of arrays.
  {1.0},                   // 1. row
  {1.0, 2.0},              // 2. row
  {1.0, 2.0, 1.0}          // 3. row
};
```

Nested loops are a natural match for manipulating multidimensional arrays. In Example 3.9, a rectangular 4 × 3 `int` matrix is declared and constructed at (1). The program finds the minimum value in the matrix. The outer loop at (2) iterates over the rows (`mXnArray[i]`, where `0` ≤ `i` < `mXnArray.length`), and the inner loop at (3) iterates over the elements in each row in turn (`mXnArray[i][j]`, where `0` ≤ `j` < `mXnArray[i].length`). The outer loop is executed `mXnArray.length` times, or four times, and the inner loop is executed (`mXnArray.length`) × (`mXnArray[i].length`), or 12 times, since all rows have the same length 3.

The `for(:)` loop also provides a safe and convenient way of iterating over an array. Several examples of its use are provided in §4.8, p. 176.

Example 3.9 *Using Multidimensional Arrays*

Click here to view code image

```
public class MultiArrays {

  public static void main(String[] args) {
    // Declare and construct the M X N matrix.
    int[][] mXnArray = {                                          // (1)
        {16,  7, 12}, // 1. row
        { 9, 20, 18}, // 2. row
        {14, 11,  5}, // 3. row
        { 8,  5, 10}  // 4. row
    }; // 4 x 3 int matrix

    // Find the minimum value in an M X N matrix:
    int min = mXnArray[0][0];
    for (int i = 0; i < mXnArray.length; ++i)                     // (2)
      // Find min in mXnArray[i], in the row given by index i:
      for (int j = 0; j < mXnArray[i].length; ++j)                // (3)
        min = Math.min(min, mXnArray[i][j]);

    System.out.println("Minimum value: " + min);
  }
}''
```

Output from the program:

```
Minimum value: 5
```

3.10 Parameter Passing

Objects communicate by calling methods on each other. A *method call* is used to invoke a method on an object. Parameters in the method call provide one way of exchanging information between the caller object and the callee object (which need not be different).

The syntax of a method call can be any one of the following:

Click here to view code image

object_reference.*method_name*(*actual_parameter_list*)

class_name.*static_method_name*(*actual_parameter_list*)

```
method_name(actual_parameter_list)
```

The *object_reference* must be an expression that evaluates to a reference value denoting the object on which the method is called. If the caller and the callee are the same, *object reference* can be omitted (see the discussion of the `this` reference on p. 106). The *class_name* can be the *fully qualified name* (§6.3, p. 326) of the class. The *actual_parameter_list* is *comma-separated* if there is more than one parameter. The parentheses are mandatory even if the actual parameter list is empty. This distinguishes the method call from field access. One can specify fully qualified names for classes and packages using the *dot operator* (`.`).

Click here to view code image

```
objRef.doIt(time, place);          // Explicit object reference
int i = java.lang.Math.abs(-1);    // Fully qualified class name
int j = Math.abs(-1);              // Simple class name
someMethod(ofValue);               // Object or class is implied
someObjRef.make().make().make();   // make() returns a reference value
```

The dot operator (`.`) has left associativity. In the last line of the preceding code, the first call of the `make()` method returns a reference value that denotes the object on which to execute the next call, and so on. This is an example of *call chaining*.

Each *actual parameter* (also called an *argument*) is an expression that is evaluated, and whose value is passed to the method when the method is invoked. Its value can vary from invocation to invocation. *Formal parameters* are parameters defined in the *method declaration* and are *local* to the method.

It should also be stressed that each invocation of a method has its own copies of the formal parameters, as is the case for any local variables in the method. The JVM uses a *stack* to keep track of method execution and a *heap* to manage the objects that are created by the program (§7.1, p. 365). Values of local variables and those passed to the method as parameters, together with any temporary values computed during the execution of the method, are always stored on the stack. Thus only primitive values and reference values are stored on the stack, and only these can be passed as parameters in a method call, but never any object from the heap.

In Java, all parameters are *passed by value*—that is, an actual parameter is evaluated and its value from the stack is assigned to the corresponding formal parameter. Table 3.2 summarizes the value that is passed depending on the type of the parameters. In the case of primitive data types, the data value of the actual parameter is passed. If the actual parameter is a reference to an object, the reference value of the denoted object is passed and not the object itself. Analogously, if the actual parameter is an array element of a primitive data type, its data value is passed, and if the array element is a reference to an object, then its reference value is passed.

Table 3.2 *Parameter Passing by Value*

Data type of the formal parameter	Value passed
Primitive data type	Primitive data value of the actual parameter
Reference type (i.e., class, interface, array, or enum type)	Reference value of the actual parameter

The order of evaluation in the actual parameter list is always *from left to right*. The evaluation of an actual parameter can be influenced by an earlier evaluation of an actual parameter. Given the following declaration:

```
int i = 4;
```

the method call

```
leftRight(i++, i);
```

is effectively the same as

```
leftRight(4, 5);
```

and not the same as

```
leftRight(4, 4);
```

An overview of the conversions that can take place in a method invocation context is provided in §2.4, p. 48. Method invocation conversions for primitive values are discussed in the next subsection (p. 129), and those for reference types are discussed in §5.10, p. 265. Calling variable arity methods is discussed in the next section (p. 136).

For the sake of simplicity, the examples in subsequent sections primarily show method invocation on the same object or the same class. The parameter passing mechanism is no different when different objects or classes are involved.

Passing Primitive Data Values

An actual parameter is an expression that is evaluated first, with the resulting value then being assigned to the corresponding formal parameter at method invocation. The use of this value in the method has no influence on the actual parameter. In particular, when the actual parameter is a variable of a primitive data type, the value of the variable from the stack is copied to the formal parameter at method invocation. Since formal parameters are local to the method, any changes made to the formal parameter will not be reflected in the actual parameter after the call completes.

Legal type conversions between actual parameters and formal parameters of *primitive data types* are summarized here from **Table 2.17**, **p. 47**:

- Primitive widening conversion
- Unboxing conversion, followed by an optional widening primitive conversion

These conversions are illustrated by invoking the following method

Click here to view code image

```
static void doIt(long i) { /* ... */ }
```

with the following code:

Click here to view code image

```
Integer intRef = 34;
Long longRef = 34L;
doIt(34);         // (1) Primitive widening conversion: long <-- int
doIt(longRef);    // (2) Unboxing: long <-- Long
doIt(intRef);     // (3) Unboxing, followed by primitive widening conversion:
                  //     long <-- int <-- Integer
```

However, for parameter passing, there are no implicit narrowing conversions for integer constant expressions (§2.4, p. 48).

Example 3.10 *Passing Primitive Values*

Click here to view code image

```
public class CustomerOne {
  public static void main (String[] args) {
    PizzaFactory pizzaHouse = new PizzaFactory();
    int pricePrPizza = 15;
    System.out.println("Value of pricePrPizza before call: " + pricePrPizza);
```

```
    double totPrice = pizzaHouse.calcPrice(4, pricePrPizza);           // (1)
    System.out.println("Value of pricePrPizza after call: " + pricePrPizza);
  }
}

class PizzaFactory {
  public double calcPrice(int numberOfPizzas, double pizzaPrice) {       // (2)
    pizzaPrice = pizzaPrice / 2.0;       // Changes price.
    System.out.println("Changed pizza price in the method: " + pizzaPrice);
    return numberOfPizzas * pizzaPrice;
  }
}
```

Output from the program:

Click here to view code image

```
Value of pricePrPizza before call: 15
Changed pizza price in the method: 7.5
Value of pricePrPizza after call: 15
```

In **Example 3.10**, the method `calcPrice()` is defined in the class `PizzaFactory` at (2). It is called from the `CustomerOne.main()` method at (1). The value of the first actual parameter, `4`, is copied to the `int` formal parameter `numberOfPizzas`. Note that the second actual parameter `pricePrPizza` is of the type `int`, while the corresponding formal parameter `pizzaPrice` is of the type `double`. Before the value of the actual parameter `pricePrPizza` is copied to the formal parameter `pizzaPrice`, it is implicitly widened to a `double`. The passing of primitive values is illustrated in **Figure 3.2**.

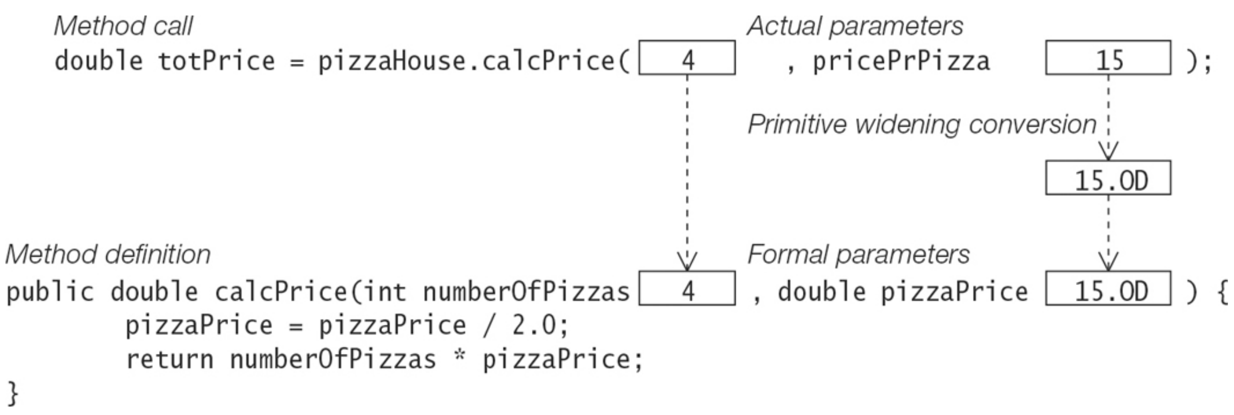

Figure 3.2 *Parameter Passing: Primitive Data Values*

The value of the formal parameter `pizzaPrice` is changed in the `calcPrice()` method, but this does not affect the value of the actual parameter `pricePrPizza` on return. It still has the value `15`. The bottom line is that the formal parameter is a local variable, and changing its value does not affect the value of the actual parameter.

Passing Reference Values

If the actual parameter expression evaluates to a reference value, the resulting reference value on the stack is assigned to the corresponding formal parameter reference at method invocation. In particular, if an actual parameter is a reference to an object, the reference value stored in the actual parameter is passed. Consequently, both the actual parameter and the formal parameter are aliases to the object denoted by this reference value during the invocation of the method. In particular, this implies that changes made to the object via the formal parameter *will* be apparent after the call returns.

Type conversions between actual and formal parameters of reference types are discussed in §5.10, p. 265.

In **Example 3.11**, a `Pizza` object is created at (1). Any object of the class `Pizza` created using the class declaration at (5) always results in a beef pizza. In the call to the `bake()` method at (2), the reference value of the object referenced by the actual parameter `favoritePizza` is assigned to the formal parameter `pizzaToBeBaked` in the declaration of the `bake()` method at (3).

Example 3.11 *Passing Reference Values*

Click here to view code image

```
public class CustomerTwo {
  public static void main (String[] args) {
    Pizza favoritePizza = new Pizza();              // (1)
    System.out.println("Meat on pizza before baking: " + favoritePizza.meat);
    bake(favoritePizza);                            // (2)
    System.out.println("Meat on pizza after baking: " + favoritePizza.meat);
  }
  public static void bake(Pizza pizzaToBeBaked) {   // (3)
    pizzaToBeBaked.meat = "chicken";  // Change the meat on the pizza.
    pizzaToBeBaked = null;                          // (4)
  }
}

class Pizza {                                       // (5)
  String meat = "beef";
}
```

Output from the program:

Click here to view code image

```
Meat on pizza before baking: beef
```

```
Meat on pizza after baking: chicken
```

One particular consequence of passing reference values to formal parameters is that any changes made to the object via formal parameters will be reflected back in the calling method when the call returns. In this case, the reference `favoritePizza` will show that chicken has been substituted for beef on the pizza. Setting the formal parameter `pizza-ToBeBaked` to `null` at (4) does not change the reference value in the actual parameter `favoritePizza`. The situation at method invocation, and just before the return from method `bake()`, is illustrated in **Figure 3.3**.

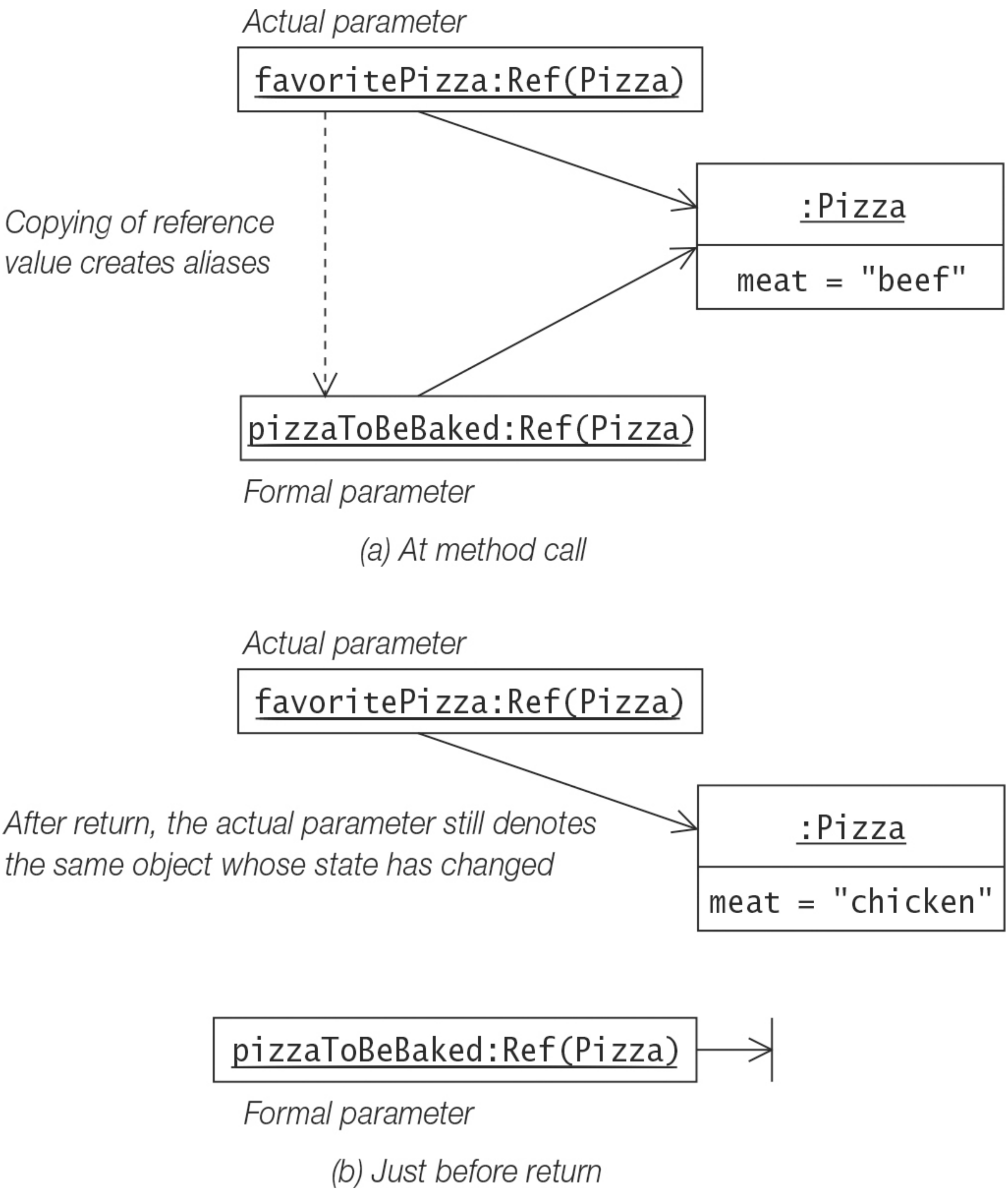

Figure 3.3 *Parameter Passing: Reference Values*

In summary, the formal parameter can only change the *state* of the object whose reference value was passed to the method.

The parameter passing strategy in Java is *call by value* and not *call by reference*, regardless of the type of the parameter. Call by reference would have allowed values in the actual parameters to be changed via formal parameters; that is, the value in `pricePrPizza` would be halved in **Example 3.10** and `favoritePizza` would be set to `null` in **Example 3.11**. However, this cannot be directly implemented in Java.

Passing Arrays

The discussion of passing reference values in the previous section is equally valid for arrays, as arrays are objects in Java. Method invocation conversions for array types are discussed along with those for other reference types in §5.10, p. 265.

In **Example 3.12**, the idea is to repeatedly swap neighboring elements in an integer array until the largest element in the array *percolates* to the last position in the array.

Example 3.12 *Passing Arrays*

Click here to view code image

```
public class Percolate {

  public static void main (String[] args) {
    int[] dataSeq = {8,4,6,2,1};    // Create and initialize an array.

    // Write array before percolation:
    printIntArray(dataSeq);

    // Percolate:
    for (int index = 1; index < dataSeq.length; ++index)

      if (dataSeq[index-1] > dataSeq[index])
        swap(dataSeq, index-1, index);                    // (1)

    // Write array after percolation:
    printIntArray(dataSeq);
  }

  public static void swap(int[] intArray, int i, int j) { // (2)
    int tmp = intArray[i]; intArray[i] = intArray[j]; intArray[j] = tmp;
  }

  public static void swap(int v1, int v2) {               // (3) Logical error!
    int tmp = v1; v1 = v2; v2 = tmp;
  }

  public static void printIntArray(int[] array) {         // (4)
    for (int value : array)
```

```
        System.out.print(" " + value);
      System.out.println();
    }
  }
```

Output from the program:

```
8 4 6 2 1
4 6 2 1 8
```

Note that in the declaration of the method `swap()` at (2), the formal parameter `intArray` is of the array type `int[]`. The other two parameters are of type `int`. They denote the values in the array that should be swapped. The signature of the method is

```
swap(int[], int, int)
```

This `swap()` method is called in the `main()` method at (1), where one of the actual parameters is the array variable `dataSeq`. The reference value of the array variable `dataSeq` is assigned to the array variable `intArray` at method invocation. After return from the call to the `swap()` method, the array variable `dataSeq` will reflect the changes made to the array via the corresponding formal parameter. This situation is depicted in **Figure 3.4** at the first call and return from the `swap()` method, indicating how the values of the elements at indices `0` and `1` in the array have been swapped.

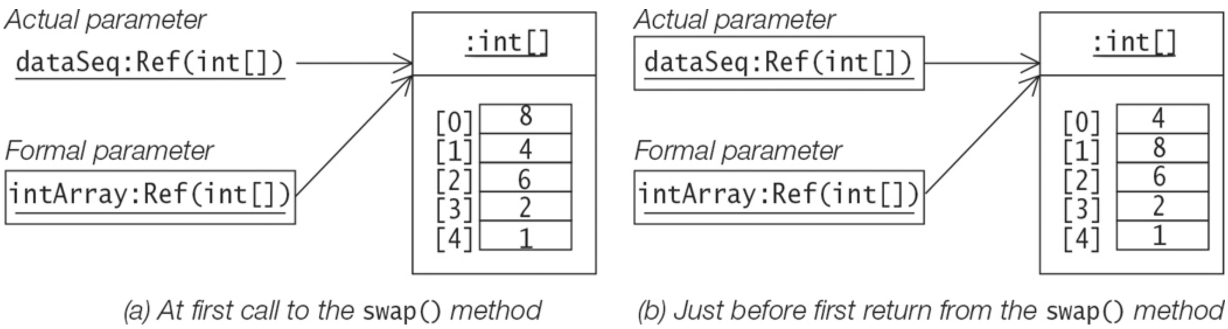

Figure 3.4 *Parameter Passing: Arrays*

However, the declaration of the `swap()` method at (3) will *not* swap two values. The method call

Click here to view code image

```
swap(dataSeq[index-1], dataSeq[index]); // Call signature: swap(int, int)
```

will result in the `swap()` method at (3) to be invoked. Its execution will have no effect on the array elements, as the swapping is done on the values of the formal parameters.

The method `printIntArray()` at (4) also has a formal parameter of array type `int[]`. Note that the formal parameter is specified as an array reference using the `[]` notation, but this notation is not used when an array is passed as an actual parameter.

Array Elements as Actual Parameters

Array elements, like other variables, can store values of primitive data types or reference values of objects. In the latter case, they can also be arrays—that is, arrays of arrays (p. 124). If an array element is of a primitive data type, its data value is passed; if it is a reference to an object, the reference value is passed. The method invocation conversions apply to the values of array elements as well.

Example 3.13 *Array Elements as Primitive Data Values*

Click here to view code image

```
public class FindMinimum {

  public static void main(String[] args) {
    int[] dataSeq = {6,4,8,2,1};

    int minValue = dataSeq[0];
    for (int index = 1; index < dataSeq.length; ++index)
      minValue = minimum(minValue, dataSeq[index]);            // (1)

    System.out.println("Minimum value: " + minValue);
  }

  public static int minimum(int i, int j) {                    // (2)
    return (i <= j) ? i : j;
  }
}
```

Output from the program:

```
Minimum value: 1
```

In Example 3.13, the value of all but one element of the array `dataSeq` is retrieved and passed consecutively at (1) to the formal parameter `j` of the `minimum()` method defined at (2). The discussion on passing primitive values (p. 129) also applies to array elements that have primitive values.

In Example 3.14, the formal parameter `seq` of the `findMinimum()` method defined at (4) is an array variable. The variable `matrix` denotes an array of arrays declared at (1) simulating a multidimensional array that has three rows, where each row is a simple array. The first row, denoted by `matrix[0]`, is passed to the `findMinimum()` method in the call at (2). Each remaining row is passed by its reference value in the call to the `findMinimum()` method at (3).

Example 3.14 *Array Elements as Reference Values*

Click here to view code image

```
public class FindMinimumMxN {

  public static void main(String[] args) {
    int[][] matrix = { {8,4},{6,3,2},{7} };                   // (1)

    int min = findMinimum(matrix[0]);                         // (2)
    for (int i = 1; i < matrix.length; ++i) {
      int minInRow = findMinimum(matrix[i]);                  // (3)
      min = Math.min(min, minInRow);
    }
    System.out.println("Minimum value in matrix: " + min);
  }

  public static int findMinimum(int[] seq) {                  // (4)
    int min = seq[0];
    for (int i = 1; i < seq.length; ++i)
      min = Math.min(min, seq[i]);
    return min;
  }
}
```

Output from the program:

```
Minimum value in matrix: 2
```

`final` Parameters

A formal parameter can be declared with the keyword `final` preceding the parameter declaration in the method declaration. A `final` parameter is also known as a *blank final variable*; that is, it is blank (uninitialized) until a value is assigned to it, (e.g., at method invocation) and then the value in the variable cannot be changed during the lifetime of the variable (see also the discussion in §6.6, p. 352). The compiler can treat `final` variables as constants for code optimization purposes. Declaring parameters as `final` prevents their values from being changed inadvertently. A formal parameter's declaration as `final` does not affect the caller's code.

The declaration of the method `calcPrice()` from **Example 3.10** is shown next, with the formal parameter `pizzaPrice` declared as `final`:

Click here to view code image

```
public double calcPrice(int numberOfPizzas, final double pizzaPrice) {  // (2')
  pizzaPrice = pizzaPrice/2.0;        // (3) Not allowed. Compile-time error!
  return numberOfPizzas * pizzaPrice;
}
```

If this declaration of the `calcPrice()` method is compiled, the compiler will not allow the value of the `final` parameter `pizzaPrice` to be changed at (3) in the body of the method.

As another example, the declaration of the method `bake()` from **Example 3.11** is shown here, with the formal parameter `pizzaToBeBaked` declared as `final`:

Click here to view code image

```
public static void bake(final Pizza pizzaToBeBaked) { // (3)
  pizzaToBeBaked.meat = "chicken";   // (3a) Allowed
  pizzaToBeBaked = null;             // (4) Not allowed. Compile-time error!
}
```

If this declaration of the `bake()` method is compiled, the compiler will not allow the reference value of the `final` parameter `pizzaToBeBaked` to be changed at (4) in the body of the method. Note that this applies to the reference value in the `final` parameter, but not to the object denoted by this parameter. The state of the object can be changed as before, as shown at (3a).

For use of the `final` keyword in other contexts, see §5.5, p. 225.

3.11 Variable Arity Methods

A *fixed arity* method must be called with the same number of actual parameters (also called *arguments*) as the number of formal parameters specified in its declaration. If the method declaration specifies two formal parameters, every call of this method must specify exactly two arguments. We say that the arity of this method is 2. In other words, the arity of such a method is fixed, and it is equal to the number of formal parameters specified in the method declaration.

Java also allows declaration of *variable arity* methods (also called *varargs* methods), meaning that the number of arguments in its call can be *varied*. As we shall see, invocations of such a method may contain more actual parameters than formal parameters.

Variable arity methods are heavily employed in formatting text representation of values, as demonstrated by the variable arity method `System.out.printf()` that is used in many examples for this purpose.

The *last* formal parameter in a variable arity method declaration is declared as follows:

Click here to view code image

```
type... formal_parameter_name
```

The ellipsis (`...`) is specified between the *type* and the *formal_parameter_name*. The *type* can be a primitive type, a reference type, or a type parameter. Whitespace can be specified on both sides of the ellipsis. Such a parameter is usually called a *variable arity parameter* (also known as a *varargs* parameter).

Apart from the variable arity parameter, a variable arity method is identical to a fixed arity method. The method `publish()` below is a variable arity method:

Click here to view code image

```
public static void publish(int n, String... data) {      // (int, String[])
  System.out.println("n: " + n + ", data size: " + data.length);
}
```

The variable arity parameter in a variable arity method is always interpreted as having an array type:

```
type[]
```

In the body of the `publish()` method, the variable arity parameter `data` has the type `String[]`, so it is a simple array of `String`s.

Only *one* variable arity parameter is permitted in the formal parameter list, and it is always the *last* parameter in the list. Given that the method declaration has *n* formal parameters and the method call has *k* actual parameters, *k* must be equal to or greater than *n* – 1. The last *k* – *n* + 1 actual parameters are evaluated and stored in an array whose reference value is passed as the value of the actual parameter. In the case of the `publish()` method, *n* is equal to 2, so *k* can be 1, 2, 3, and so on. The following invocations of the `publish()` method show which arguments are passed in each method call:

Click here to view code image

```
publish(1);                   // (1, new String[] {})
publish(2, "two");            // (2, new String[] {"two"})
publish(3, "two", "three");  // (3, new String[] {"two", "three"})
```

Each method call results in an *implicit* array being created and passed as an argument. This array can contain zero or more argument values that do *not* correspond to the formal parameters preceding the variable arity parameter. This array is referenced by the variable arity parameter `data` in the method declaration. The preceding calls would result in the `publish()` method printing the following output:

```
n: 1, data size: 0
n: 2, data size: 1
n: 3, data size: 2
```

To overload a variable arity method, it is not enough to change the type of the variable arity parameter to an explicit array type. The compiler will complain if an attempt is made to overload the method `transmit()`, as shown in the following code:

Click here to view code image

```
public static void transmit(String... data) {  }  // Compile-time error!
public static void transmit(String[] data)  {  }  // Compile-time error!
```

Both methods above have the signature `transmit(String[])`. These declarations would result in two methods with equivalent signatures in the same class, which is not permitted.

Overloading and overriding of methods with variable arity are discussed in §5.10, p. 265.

Calling a Variable Arity Method

Example 3.15 illustrates various aspects of calling a variable arity method. The method `flexiPrint()` in the `VarargsDemo` class has a variable arity parameter:

Click here to view code image

```
public static void flexiPrint(Object... data) { // Object[]
  //...
}
```

The variable arity method prints the name of the `Class` object representing the *actual array* that is passed at runtime. It prints the number of elements in this array as well as the text representation of each element in the array.

The method `flexiPrint()` is called in the `main()` method. First it is called with the values of primitive types and `String` s ((1) to (8)), and then it is called with the program arguments (p. 141) supplied on the command line ((9) to (11)).

Compiling the program results in a *warning* at (9), which we ignore for the time being. The program can still be run, as shown in Example 3.15. The numbers at the end of the lines in the output relate to numbers in the code, and are not printed by the program.

Example 3.15 *Calling a Variable Arity Method*

Click here to view code image

```
public class VarargsDemo {
  public static void flexiPrint(Object... data) { // Object[]
    // Print the name of the Class object for the varargs parameter.
    System.out.print("Type: " + data.getClass().getName());

    System.out.println("  No. of elements: " + data.length);

    System.out.print("Element values: ");
    for(Object element : data)
      System.out.print(element + " ");
    System.out.println();
  }

  public static void main(String... args) {
    int    day       = 13;
    String monthName = "August";
    int    year      = 2009;

    // Passing primitives and non-array types:
    flexiPrint();                     // (1) new Object[] {}
    flexiPrint(day);                  // (2) new Object[] {Integer.valueOf(day)}
    flexiPrint(day, monthName);       // (3) new Object[] {Integer.valueOf(day),
                                      //                   monthName}
    flexiPrint(day, monthName, year); // (4) new Object[] {Integer.valueOf(day),
                                      //                   monthName,
                                      //                   Integer.valueOf(year)}
    System.out.println();

    // Passing an array type:
    Object[] dateInfo = {day,         // (5) new Object[] {Integer.valueOf(day),
                         monthName,   //                   monthName,
                         year};       //                   Integer.valueOf(year)}
    flexiPrint(dateInfo);             // (6) Non-varargs call
    flexiPrint((Object) dateInfo);    // (7) new Object[] {(Object) dateInfo}
    flexiPrint(new Object[]{dateInfo});// (8) Non-varargs call
    System.out.println();
```

```
    // Explicit varargs or non-varargs call:
    flexiPrint(args);                  // (9) Warning!
    flexiPrint((Object) args);         // (10) Explicit varargs call
    flexiPrint((Object[]) args);       // (11) Explicit non-varargs call
  }
}
```

Compiling the program:

Click here to view code image

```
>javac VarargsDemo.java
VarargsDemo.java:41: warning: non-varargs call of varargs method with inexact
argument type for last parameter;
    flexiPrint(args);                  // (9) Warning!
               ^
  cast to Object for a varargs call
  cast to Object[] for a non-varargs call and to suppress this warning
1 warning
```

Running the program:

Click here to view code image

```
>java VarargsDemo To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 0                (1)
Element values:
Type: [Ljava.lang.Object;  No. of elements: 1                (2)
Element values: 13
Type: [Ljava.lang.Object;  No. of elements: 2                (3)
Element values: 13 August
Type: [Ljava.lang.Object;  No. of elements: 3                (4)
Element values: 13 August 2009

Type: [Ljava.lang.Object;  No. of elements: 3                (6)
Element values: 13 August 2009
Type: [Ljava.lang.Object;  No. of elements: 1                (7)
Element values: [Ljava.lang.Object;@1eed786
Type: [Ljava.lang.Object;  No. of elements: 1                (8)
Element values: [Ljava.lang.Object;@1eed786

Type: [Ljava.lang.String;  No. of elements: 6                (9)
Element values: To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 1                (10)
Element values: [Ljava.lang.String;@187aeca
Type: [Ljava.lang.String;  No. of elements: 6                (11)
Element values: To arg or not to arg
```

Variable Arity and Fixed Arity Method Calls

The calls at (1) to (4) in **Example 3.15** are all *variable arity calls*, as an implicit `Object` array is created in which the values of the actual parameters are stored. The reference value of this array is passed to the method. The printout shows that the type of the parameter is actually an array of `Object` s (`[Ljava.lang.Object;`).

The call at (6) differs from the previous calls in that the actual parameter is an array that has the *same* type (`Object[]`) as the variable arity parameter, without having to create an implicit array. In such a case, *no* implicit array is created, and the reference value of the array `dateInfo` is passed to the method. See also the result from this call at (6) in the output. The call at (6) is a *fixed arity call* (also called a *non-varargs call*), where no implicit array is created:

Click here to view code image

```
flexiPrint(dateInfo);              // (6) Non-varargs call
```

However, if the actual parameter is cast to the type `Object` as at (7), a *variable arity* call is executed:

Click here to view code image

```
flexiPrint((Object) dateInfo);     // (7) new Object[] {(Object) dateInfo}
```

The type of the actual argument (`Object`) is now *not* the same as that of the variable arity parameter (`Object[]`), resulting in an array of the type `Object[]` being created in which the array `dateInfo` is stored as an element. The printout at (7) shows that only the text representation of the `dateInfo` array is printed, and not its elements, as it is the sole element of the implicit array.

The call at (8) is a *fixed arity* call, for the same reason as the call at (6). Now, however, the array `dateInfo` is explicitly stored as an element in an array of the type `Object[]` that matches the type of the variable arity parameter:

Click here to view code image

```
flexiPrint(new Object[]{dateInfo});// (8) Non-varargs call
```

The output from (8) is the same as the output from (7), where the array `dateInfo` was passed as an element in an implicitly created array of type `Object[]`.

The compiler issues a *warning* for the call at (9):

Click here to view code image

```
flexiPrint(args);                    // (9) Warning!
```

The actual parameter `args` is an array of the type `String[]`, which is a *subtype* of `Object[]`—the type of the variable arity parameter. The array `args` can be passed in a fixed arity call as an array of the type `String[]`, or in a variable arity call as *an element* in an implicitly created array of the type `Object[]`. *Both* calls are feasible and valid in this case. Note that the compiler chooses a fixed arity call rather than a variable arity call, but also issues a warning. The result at (9) confirms this course of action. A warning at compile time is not the same as a compile-time error. The former does not prevent the program from being run, whereas the latter does.

At (10), the array `args` of the type `String[]` is explicitly passed as an `Object` in a variable arity call, similar to the call at (7):

Click here to view code image

```
flexiPrint((Object) args);           // (10) Explicit varargs call
```

At (11), the array `args` of type `String[]` is explicitly passed as an array of the type `Object[]` in a fixed arity call. This call is equivalent to the call at (9), where the widening reference conversion is implicit, but now without a warning at compile time. The two calls print the same information, as is evident from the output at (9) and (11):

Click here to view code image

```
flexiPrint((Object[]) args);         // (11) Explicit non-varargs call
```

3.12 The `main()` Method

The mechanics of compiling and running Java applications using the JDK are outlined in §1.8, p. 19. The `java` command executes a method called `main` in the class specified on the command line. This class designates the *entry point of the application*. Any class can have a `main()` method, but only the `main()` method of the class specified in the `java` command starts the execution of a Java application.

The `main()` method must have `public` access so that the JVM can call this method (§6.5, p. 345). It is a `static` method belonging to the class, so no object of the class is required to start its execution. It does not return a value; that is, it is declared as `void`. It has an array of `String` objects as its only formal parameter. This array contains any arguments passed to the program on the command line (see the next subsection). The following method header declarations fit the bill, and any one of them can be used for the `main()` method:

Click here to view code image

```
public static void main(String[] args)    // Method header
public static void main(String args[])    // Method header
public static void main(String... args)   // Method header
```

The three modifiers can occur in any order in the method header. The requirements given in these examples do not exclude specification of other non-access modifiers like `final` (§5.5, p. 226) or a `throws` clause (§7.5, p. 388). The `main()` method can also be overloaded like any other method. The JVM ensures that the `main()` method having the correct method header is the starting point of program execution.

Program Arguments

Any arguments passed to the program on the command line can be accessed in the `main()` method of the class specified on the command line. These arguments are passed to the `main()` method via its formal parameter `args` of type `String[]`. These arguments are called *program arguments*.

In **Example 3.16**, the program prints the arguments passed to the `main()` method from the following command line:

Click here to view code image

```
>java Colors red yellow green "blue velvet"
```

The program prints the total number of arguments given by the field `length` of the `String` array `args`. Each string in `args`, which corresponds to a program argument, is printed together with its length inside a `for` loop. From the output, we see that there are four program arguments. On the command line, the arguments can be separated by one or more spaces between them, but these are not part of any argument. The last argument shows that we can quote the argument if spaces are to be included as part of the argument.

When no arguments are specified on the command line, a `String` array of zero length is created and passed to the `main()` method. Thus the reference value of the formal parameter in the `main()` method is never `null`.

Note that the command name `java` and the class name `Colors` are not passed to the `main()` method of the class `Colors`, nor are any other options that are specified on the command line.

As program arguments can only be passed as strings, they must be explicitly converted to other values by the program, if necessary.

Program arguments supply information to the application, which can be used to tailor the runtime behavior of the application according to user requirements.

Example 3.16 *Passing Program Arguments*

Click here to view code image

```
public class Colors {
   synchronized public static void main(String[] args) {
    System.out.println("No. of program arguments: " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println("Argument no. " + i + " (" + args[i] + ") has " +
                              args[i].length() + " characters.");
  }
}
```

Running the program:

Click here to view code image

```
>java Colors red yellow green "blue velvet"
No. of program arguments: 4
Argument no. 0 (red) has 3 characters.
Argument no. 1 (yellow) has 6 characters.
Argument no. 2 (green) has 5 characters.
Argument no. 3 (blue velvet) has 11 characters.
```

3.13 Local Variable Type Inference

A variable declaration requires the *type* of the variable to be specified in the declaration. However, in the case of *local variables*, the type can be specified by the reserved type name `var`, if the local declaration also specifies an *initialization expression* in the declaration. The compiler uses the type of the initialization expression to *infer* the type of the local variable. The restricted type name `var` denotes this inferred type in the local declaration. This is an example of *type inference*, where the type of a variable or an expression is derived from the context in which it is used. If the compiler cannot infer the type, it reports a compile-time error. A local variable declaration that uses `var` is also called a `var` *declaration*. A local variable declared this way is no different from any other local variable.

It is important to note that the type of the local variable is *solely* inferred from the initialization expression specified in the declaration. The following variable declaration in a local context (e.g., body of a method) is declared using the reserved type name `var`:

```
var year = 2022;
```

The compiler is able to infer that the type of the initialization expression `2022` is `int` in the above declaration, and therefore the variable `year` has the type `int`. The declaration above is equivalent to the declaration below, where the type is explicitly specified:

```
int year = 2022;
```

A cautionary note going forward: This subsection refers to many concepts and constructs that might not be familiar at this stage. It might be a good idea to get an overview now and to come back later for a more thorough review of this topic. The exhaustive index at the end of the book can of course be used at any time to look up a topic.

The class `ValidLVTI` in **Example 3.17** illustrates valid uses of the restricted type name `var`. The comments in the code should be self-explanatory.

The `var` restricted type name is allowed in local variable declarations in *blocks* (including *initializer blocks*), *constructors*, and *methods*, as can be seen in the class `ValidLVTI` at (1a), (1b), and (2) and the method `main()`, respectively.

Note that at (3b) and (3c), the compiler is able to infer the type of the local variable from the return type of the method on the right-hand side.

It is worth noting that the *cast operator*, `()`, can be necessary to indicate the desired type, as shown at (5) and (7).

For array variables, the initialization expression must be an *array creation expression* that allows the array size and the array element type to be inferred, as shown at (11a), (11b), (11c), and (11d). A local declaration with `var` requires an initialization expression, which in the case of local arrays must be either an array creation expression or an anonymous array expression. In other words, it should be possible to infer both the array element type and the size of the array. It cannot be an array initializer.

The bodies (and the headers) of the `for(;;)` and `for(:)` *loops* can define their own local variables in their *block scope*. The type of the local variable `vowel` at (13) is inferred to be `char` from the array `vowels` (of type `char[]`) in the header of the `for(:)` loop. The type of the local variable `i` in the header of the `for(;;)` loop at (16) is determined to be `int` from the initial value. The `switch` statement also defines its own block scope in which local variables can be declared, as shown at (18).

Example 3.17 *Illustrating Local Variable Type Reference*

Click here to view code image

```
// Class ValidLVTI illustrates valid use of the restricted type name var.
public class ValidLVTI {

  // Static initializer block:
  static {
    var slogan = "Keep calm and code Java.";        // (1a) Allowed in static
  }                                                 //      initializer block

  // Instance initializer block:
  {
    var banner = "Keep calm and catch exceptions."; // (1b) Allowed in instance
  }                                                 //      initializer block

  // Constructor:
  public ValidLVTI() {
    var luckyNumber = 13;                       // (2) Allowed in a constructor.
  }

  // Method:
  public static void main(String[] args) {

    var virus = "COVID-19";                     // (3a) Type of virus is String.
    var acronym = virus.substring(0, 5);        // (3b) Type of acronym is String.
    var num = Integer.parseInt(virus.substring(6)); // (3c) Type of num is int.

    var obj = new Object();                     // (4) Type of obj is Object.
    var title = (String) null; // (5) Initialization expression type is String.
                               //     Type of title is String.
    var sqrtOfNumber = Math.sqrt(100); // (6) Type of sqrtOfNumber is double,
                                       //     since the method returns
                                       //     a double value.

    var tvSize  = (short) 55;  // (7) Type of tvSize is short.
    var tvSize2 = 65;          // (8) Type of tvSize2 is int.

    var diameter = 10.0;       // (9) Type of diameter is double.
    var radius = 2.5F;         // (10) Type of radius is float.

    // Arrays:
    var vowels = new char[] {'a', 'e', 'i', 'o', 'u' }; // (11a) Type of vowels
                                                        // is char[]. Size is 5.
    var zodiacSigns = new String[12]; // (11b) Type of zodiacSigns is String[].
                                      //       Size is 12.
    var a_2x3 = new int[2][3]; // (11c) Type of a_2x3 is int[][]. Size is 2x3.
    var a_2xn = new int[2][];  // (11d) Type of a_2xn is int[][]. Size is 2x?,
```

```
                                  //       where second dimension can be undefined.

      // The for(:) loop:
      var word1 = "";              // (12) Type of word2 is String.
      for (var vowel : vowels) { // (13) Type of vowel is char in the for(:)loop.
        var letter = vowel;        // (14) Type of letter is char.
        word1 += letter;
      }

      // The for(;;) loop:
      var word2 = "";                                // (15) Type of word2 is String.
      for (var i = 0; i < vowels.length; i++) { // (16) Type of i is int in
                                                     //      the for loop.
        var letter = vowels[i];                      // (17) Type of letter is char.
        word2 += letter;
      }

      // switch-statement:
      switch(virus) {
        case "Covid-19":
          var flag = "Needs to be tested.";      // (18) Type is String.
          // Do testing.
          break;
        default: // Do nothing.
      }
    }
  }
```

Click here to view code image

```
// Class InvalidLVTI illustrates invalid use of the restricted type name var.
public class InvalidLVTI {

  var javaVendor = "Oracle"; // (19) Not allowed in instance variable declaration.

  static var javaVersion = 11; // (20) Not allowed in static variable declaration.

  public static void main(var args) { // (21) Not allowed for method parameters.

    var name;              // (22) Not allowed without initialization expression.

    var objRef = null;     // (23) Literal null not allowed.

    var x = 10.0, y = 20.0, z = 40;   // (24) Not allowed in compound declaration.

    var vowelsOnly = {'a', 'e', 'i', 'o', 'u' }; // (25) Array initializer not
                                                  //      allowed.
    var attendance = new int[];          // (26) Non-empty dimension required.
    var array3Dim = new String[][2][];  // (27) Cannot specify an empty dimension
```

```
                                                    //       before a non-empty dimension.
      var letters[] = new char[]{'a', 'e', 'i', 'o', 'u' }; // (28) var not allowed
                                                            //       as element type.

      var prompt = prompt + 1;             // (29) Self-reference not allowed in
                                           //       initialization expression.
    }

    public static var getPlatformName() { // (30) Not allowed as return type.
      return "JDK";
    }
  }
```

The following examples of invalid uses of the restricted type name `var` are shown in the class `InvalidLVTI` in **Example 3.17**:

- *Not allowed in field variable declarations*
  The `var` restricted type name is not allowed in field variable declarations, as shown at (19) and (20).
- *Not allowed in declaring formal parameters*
  Formal parameters in methods and constructors cannot be declared with `var`, as shown at (21) for the parameter `args` in the `main()` method.
- *Initialization expression is mandatory*
  The `var` restricted type name is not allowed in a local variable declaration if an initialization expression is not specified, as shown at (22).
- *Initialization expression cannot be the* `null` *literal value*
  Since the literal `null` can be assigned to any reference type, a specific type for `objRef` at (23) cannot be determined. At (5), the cast `(String)` specifies the type of the initialization expression.
- *Cannot use* `var` *in compound declarations*
  The reserved type name `var` cannot be used in a *compound declaration*—that is, a declaration that declares several variables, as shown at (24).
- *Cannot use* `var` *when an array initializer is specified*
  As shown at (25), an *array initializer* cannot be used in a `var` declaration. However, an *array initialization expression* is allowed, as at (11a).
- *Array creation expression must specify the size*
  As in the case when an explicit type is specified for an array variable, the array creation expressions in the declaration must also specify the array size when using `var`; otherwise, the compiler will issue an error, as at (26) and (27). Valid array creation expressions specifying correct size are shown at (11b), (11c), and (11d).
- *Cannot use* `var` *as an array element type*
  The square brackets (`[]`) on the left-hand side at (28) are not allowed, as they indicate that the local variable is an array. Array type and size are solely determined from the initialization expression, as at (11a), (11b), (11c), and (11d).

- *Cannot have a self-reference in an initialization expression*
  As in the case when an explicit type is specified for the local variable, the initialization expression cannot refer to the local variable being declared, as at (29), where the variable is not initialized before use.
- *Cannot use* `var` *as the return type of a method*
  The method declaration at (30) cannot specify the return type using `var`.
- *A type cannot be a named* `var`
  As `var` is a reserved type name, it is *not* a valid name for a reference type; that is, a class, an interface, or an enum cannot be named `var`. In other contexts, it can be used as an identifier, but this is not recommended.
  Click here to view code image

  ```
  public class var {}   // var is not permitted as a class name. Compile-time error!
  ```

The reserved type name `var` should be used judiciously as the code can become difficult to understand. When reading the local declaration below, the initialization expression does not divulge any information about the type, and the names are not too helpful:

```
var x = gizmo.get();
```

Unless it is intuitively obvious, a human reader will have to resort to the API documentation in order to infer the type. Using intuitive names becomes even more important when using the reserved type name `var`.

We will revisit the restricted type name `var` when discussing *exception handling with try-with-resources* (§7.7, p. 407), using *generics* in local variable declarations (§11.2, p. 571), and specifying *inferred-type lambda parameters* (§13.2, p. 680).

Review Questions

3.7 How many of the following array declaration statements are legal?

Click here to view code image

```
int []aa[] = new int [4][4];
int bb[][] = new int [4][4];
int cc[][] = new int [][4];
int []dd[] = new int [4][];
int [][]ee = new int [4][4];
```

Select the one correct answer.

a. 0

b. 1

c. 2

d. 3

e. 4

f. 5

3.8 Which of these array declaration statements are legal? Select the three correct answers.

a. `int[] i[] = { { 1, 2 }, { 1 }, {}, { 1, 2, 3 } };`

b. `int i[] = new int[2] {1, 2};`

c. `int i[][] = new int[][] { {1, 2, 3}, {4, 5, 6} };`

d. `int[][] i = { { 1, 2 }, new int[2] };`

e. `int i[4] = { 1, 2, 3, 4 };`

3.9 What would be the result of compiling and running the following program?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    int size = 20;
    int[] arr = new int[ size ];
    for (int i = 0; i < size; ++i) {
      System.out.println(arr[i]);
    }
  }
}
```

Select the one correct answer.

a. The code will fail to compile because the array type `int[]` is incorrect.

b. The program will compile, but it will throw an `ArrayIndexOutOfBoundsException` when run.

c. The program will compile and run without error, but it will produce no output.

d. The program will compile and run without error and will print the numbers 0 through 19.

e. The program will compile and run without error and will print `0` twenty times.

f. The program will compile and run without error and will print `null` twenty times.

3.10 What would be the result of compiling and running the following program?

Click here to view code image

```
public class DefaultValuesTest {
  int[] ia = new int[1];
  boolean b;
  int i;
  Object o;

  public static void main(String[] args) {
    DefaultValuesTest instance = new DefaultValuesTest();
    instance.print();
  }

  public void print() {
    System.out.println(ia[0] + " " + b + " " + i + " " + o);
  }
}
```

Select the one correct answer.

a. The program will fail to compile because of uninitialized variables.

b. The program will throw a `java.lang.NullPointerException` when run.

c. The program will print `0 false NaN null`.

d. The program will print `0 false 0 null`.

e. The program will print `null 0 0 null`.

f. The program will print `null false 0 null`.

3.11 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class ParameterPass {
  public static void main(String[] args) {
    int i = 0;
    addTwo(i++);
    System.out.println(i);
  }
  static void addTwo(int i) {
    i += 2;
  }
}
```

Select the one correct answer.

a. 0

b. 1

c. 2

d. 3

3.12 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Passing {
  public static void main(String[] args) {
    int a = 0; int b = 0;
    int[] bArr = new int[1]; bArr[0] = b;

    inc1(a); inc2(bArr);

    System.out.println("a=" + a + " b=" + b + " bArr[0]=" + bArr[0]);
  }

  public static void inc1(int x) { x++; }

  public static void inc2(int[] x) { x[0]++; }
}
```

Select the one correct answer.

a. The code will fail to compile, since `x[0]++;` is not a legal statement.

b. The code will compile and will print `a=1 b=1 bArr[0]=1` at runtime.

c. The code will compile and will print `a=0 b=1 bArr[0]=1` at runtime.

d. The code will compile and will print `a=0 b=0 bArr[0]=1` at runtime.

e. The code will compile and will print `a=0 b=0 bArr[0]=0` at runtime.

3.13 Given the following code:

Click here to view code image

```
public class RQ810A40 {
  static void print(Object... obj) {
    System.out.println("Object...: " + obj[0]);
  }
  public static void main(String[] args) {
    // (1) INSERT METHOD CALL HERE
  }
}
```

Which method call, when inserted at (1), will not result in the following output from the program?

```
Object...: 9
```

Select the one correct answer.

a. `print("9", "1", "1");`

b. `print(9, 1, 1);`

c. `print(new int[] {9, 1, 1});`

d. `print(new Integer[] {9, 1, 1});`

e. `print(new String[] {"9", "1", "1"});`

f. `print(new Object[] {"9", "1", "1"});`

g. None of the above

3.14 Which statement is true about the following program?

Click here to view code image

```
public class Test {
  public static int add(int x, int y)  {
    var z = x + y;     // (1)
    return z;
  }
  public static void main(String[] args) {
    var a = 2, b = 3;  // (2)
    var z = add(a,b);  // (3)
  }
}
```

Select the one correct answer.

a. Line (1) will fail to compile.

b. Line (2) will fail to compile.

c. Line (3) will fail to compile.

d. The code will compile successfully.

3.15 Given the following code:

Click here to view code image

```
public class TestMe {
  /* (1) INSERT METHOD HEADER HERE */ {
    return (double)x / y;
  }
  public static void main(String[] args) {
    double x = divide(2, 3);
  }
}
```

Which method headers, when inserted individually at (1), will allow the code to compile?

Select the three correct answers.

a. `public static var divide(double x, double y)`

b. `public static double divide(double x, var y)`

c. `public static double divide(var x, double y)`

d. `public static double divide(int x, int y)`

e. `public static double divide(int x, double y)`

f. `public static double divide(double x, int y)`