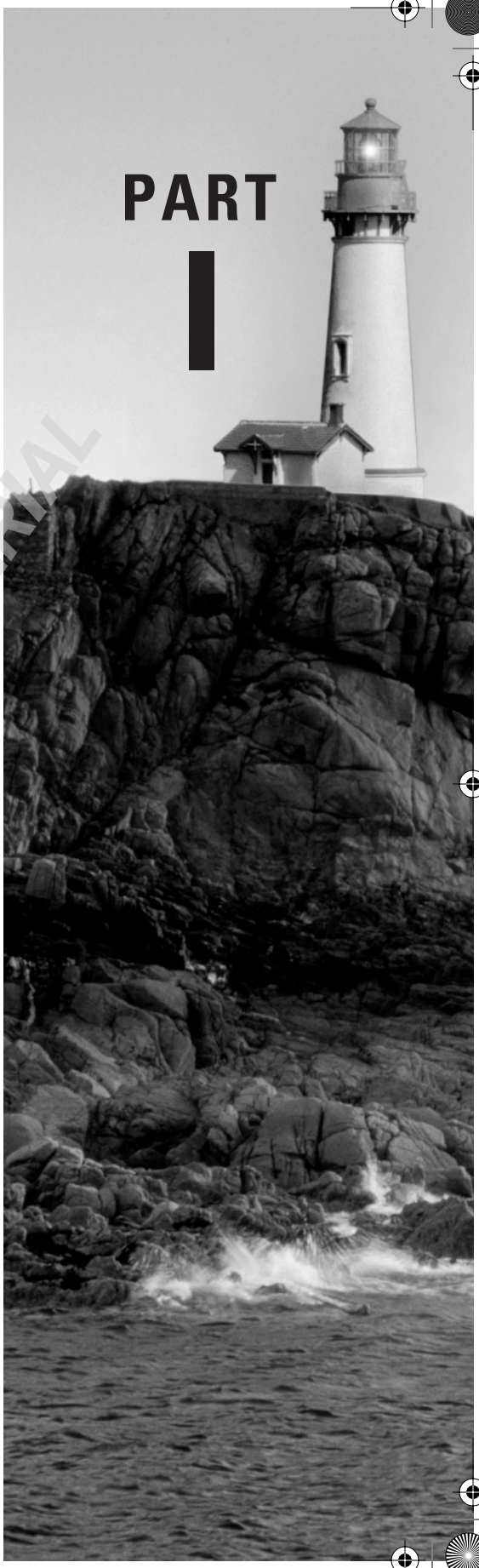
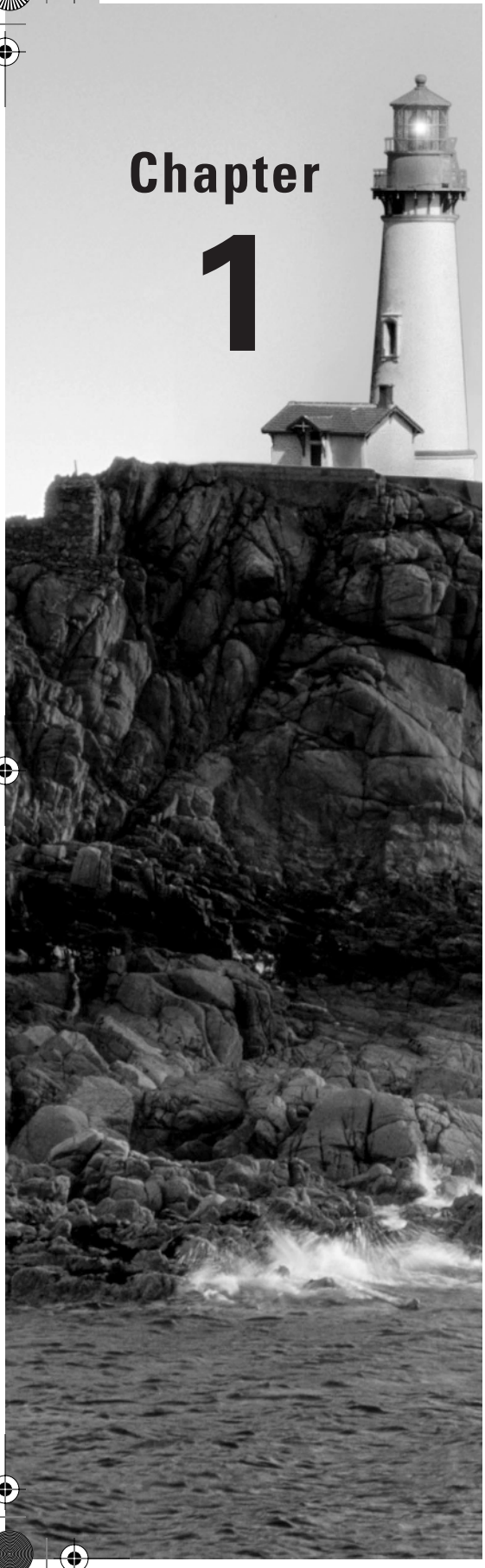
# PART I

# The Sun Certified Java Programmer Exam





Chapter 1

# Language Fundamentals

## JAVA CERTIFICATION EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports).**
- ✓ **1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.**
- ✓ **7.2 Given an example of a class and a command-line, determine the expected runtime behavior.**
- ✓ **7.3 Determine the effect upon object references and primitive values when they are passed into methods that perform assignments or other modifying operations on the parameters.**
- ✓ **7.4 Given a code example, recognize the point at which an object becomes eligible for garbage collection, and determine what is and is not guaranteed by the garbage collection system. Recognize the behaviors of System.gc and finalization.**

This book is not an introduction to Java. Since you're getting ready to take the Programmer Exam, it's safe to assume that you know how to write code, what an object is, what a constructor is, and so on. So we're going to dive right in and start looking at what you need to know to pass the exam.

This chapter covers a lot of objectives. They may seem unrelated, but they all have a common thread: they deal with the fundamentals of the language. Here you will look at Java's keywords and identifiers. Then you'll read about primitive data types and the literal values that can be assigned to them. You'll also cover some vital information about arrays, variable initialization, argument passing, and garbage collection.

# Source Files

All Java source files must end with the `.java` extension. A source file should generally contain, at most, one top-level public class definition; if a public class is present, the class name should match the unextended filename. For example, if a source file contains a public class called `RayTraceApplet`, then the file must be called `RayTraceApplet.java`. A source file may contain an unlimited number of non-public class definitions.


This is not actually a language requirement, but it is an implementation requirement of many compilers, including the reference compilers from Sun. It is unwise to ignore this convention, because doing so limits the portability of your source files (but not, of course, your compiled files).

Three top-level elements known as *compilation units* may appear in a file. None of these elements is required. If they are present, then they must appear in the following order:

1. Package declaration
2. Import statements
3. Class, interface, and enum definitions

The format of the package declaration is quite simple. The keyword `package` occurs first and is followed by the package name. The package name is a series of elements separated by periods. When class files are created, they must be placed in a directory hierarchy that reflects their package names. You must be careful that each component of your package name hierarchy is a legitimate

directory name on all platforms. Therefore, you must not use characters such as the space, forward slash, backslash, or other symbols. Use only alphanumeric characters in package names.

Import statements have a similar form, but you may import either an individual class from a package or the entire package. To import an individual class, simply place the fully qualified class name after the `import` keyword and finish the statement with a semicolon (;); to import an entire package, simply add an asterisk (*) to the end of the package name.

Java's import functionality was enhanced in 5.0. For more information, see the "Importing" section later in this chapter.

White space and comments may appear before or after any of these elements.

For example, a file called `Test.java` might look like this:

```
1. // Package declaration
2. package exam.prepguide;
3.
4. // Imports
5. import java.awt.Button; // imports a specific class
6. import java.util.*;     // imports an entire package
7.
8. // Class definition
9. public class Test {...}
```

Sometimes you might use classes with the same name in two different packages, such as the `Date` classes in the packages `java.util` and `java.sql`. If you use the asterisk form of import to import both entire packages and then attempt to use a class simply called `Date`, you will get a compiler error reporting that this usage is ambiguous. You must either make an additional import, naming one or the other `Date` class explicitly, or you must refer to the class using its fully qualified name.

# Keywords and Identifiers

A *keyword* is a word whose meaning is defined by the programming language. Anyone who claims to be competent in a language must at the very least be familiar with that language's keywords. Java's keywords and other special-meaning words are listed in Table 1.1.

Most of the words in Table 1.1 are keywords. Strictly speaking, `true` and `false` aren't really keywords, they are literal boolean values. Also, `goto` and `const` are *reserved words*, which means that although they have no meaning to the Java compiler, programmers may not use them as identifiers.

**TABLE 1.1** Java Keywords and Reserved Words

| | | | | | | |
|---|---|---|---|---|---|---|
| abstract | class | extends | implements | null | strictfp | true |
| assert | const | false | import | package | super | try |
| boolean | continue | final | instanceof | private | switch | void |
| break | default | finally | int | protected | synchronized | volatile |
| byte | do | float | interface | public | this | while |
| case | double | for | long | return | throw | |
| catch | else | goto | native | short | throws | |
| char | enum | if | new | static | transient | |

Fortunately, the exam doesn't require you to distinguish among keywords, literal booleans, and reserved words. You won't be asked trick questions like "Is `goto` a keyword?" You *will* be expected to know what each word in Table 1.1 does, except for `strictfp`, `transient`, and `volatile`.

An *identifier* is a word used by a programmer to name a variable, method, class, or label. Keywords and reserved words may not be used as identifiers. An identifier must begin with a letter, a dollar sign ($), or an underscore (_); subsequent characters may be letters, dollar signs, underscores, or digits.

Some examples are

```
foobar                // legal
BIGinterface          // legal: embedded keywords are ok
$incomeAfterTaxes     // legal
3_node5               // illegal: starts with a digit
!theCase              // illegal: bad 1st char
```

Identifiers are case sensitive—for example, `radius` and `Radius` are distinct identifiers.

The exam is careful to avoid potentially ambiguous questions that require you to make purely academic distinctions between reserved words and keywords.

# Primitive Data Types

A *primitive* is a simple non-object data type that represents a single value. Java's primitive data types are

- `boolean`
- `char`
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`

The apparent bit patterns of these types are defined in the Java language specification, and their effective sizes are listed in Table 1.2.


Variables of type `boolean` may take only the values `true` or `false`. Their representation size might vary.

**TABLE 1.2** Primitive Data Types and Their Effective Sizes

| Type | Effective Representation Size (bits) |
|---|---|
| `byte` | 8 |
| `int` | 32 |
| `float` | 32 |
| `char` | 16 |
| `short` | 16 |
| `long` | 64 |
| `double` | 64 |

A *signed data type* is a numeric type whose value can be positive, zero, or negative. (So the number has an implicit plus *sign* or minus *sign*.) An *unsigned data type* is a numeric type whose value can only be positive or zero. The four signed integral data types are

- `byte`
- `short`
- `int`
- `long`

Variables of these types are two's-complement numbers; their ranges are given in Table 1.3. Notice that for each type, the exponent of 2 in the minimum and maximum is one less than the size of the type.


Two's-complement is a way of representing signed integers that was originally developed for microprocessors in such a way as to have a single binary representation for the number 0. The most significant bit is used as the sign bit, where 0 is positive and 1 is negative.

**TABLE 1.3** Ranges of the Integral Primitive Types

| Type | Size | Minimum | Maximum |
|---|---|---|---|
| `byte` | 8 bits | $-2^7$ | $2^7 - 1$ |
| `short` | 16 bits | $-2^{15}$ | $2^{15} - 1$ |
| `int` | 32 bits | $-2^{31}$ | $2^{31} - 1$ |
| `long` | 64 bits | $-2^{63}$ | $2^{63} - 1$ |

The `char` type is integral but unsigned. The range of a variable of type `char` is from 0 through $2^{16} - 1$. Java characters are in Unicode, which is a 16-bit encoding capable of representing a wide range of international characters. If the most significant 9 bits of a `char` are all 0, then the encoding is the same as 7-bit ASCII.

The two floating-point types are

- `float`
- `double`

The ranges of the floating-point primitive types are given in Table 1.4.

**TABLE 1.4** Ranges of the Floating-Point Primitive Types

| Type | Size | Minimum | Maximum |
|---|---|---|---|
| `float` | 32 bits | +/–1.40239846$^{-45}$ | +/–3.40282347$^{+38}$ |
| `double` | 64 bits | +/–4.94065645841246544$^{-324}$ | +/–1.79769313486231570$^{+308}$ |

These types conform to the IEEE 754 specification. Many mathematical operations can yield results that have no expression in numbers (infinity, for example). To describe such non-numeric situations, both `double` and `float` can take on values that are bit patterns that do not represent numbers. Rather, these patterns represent non-numeric values. The patterns are defined in the `Float` and `Double` classes and may be referenced as follows (NaN stands for Not a Number):

- `Float.NaN`
- `Float.NEGATIVE_INFINITY`
- `Float.POSITIVE_INFINITY`
- `Double.NaN`
- `Double.NEGATIVE_INFINITY`
- `Double.POSITIVE_INFINITY`

The following code fragment shows the use of these constants:

```
1. double d = -10.0 / 0.0;
2. if (d == Double.NEGATIVE_INFINITY) {
3.   System.out.println("d just exploded: " + d);
4. }
```

In this code fragment, the test on line 2 passes, so line 3 is executed.


All numeric primitive types are signed.

# Literals

A *literal* is a value specified in the program source, as opposed to one determined at runtime. Literals can represent primitive or string variables and may appear on the right side of assignments or in method calls. You cannot assign values into literals, so they cannot appear on the left side of assignments.

In this section you'll look at the literal values that can be assigned to boolean, character, integer, floating-point, and String variables.

The only valid literals of `boolean` type are `true` and `false`. For example:

```
1. boolean isBig = true;
2. boolean isLittle = false;
```

A *chararacter literal* (`char`) represents a single Unicode character. (*Unicode* is a convention for using 16-bit unsigned numeric values to represent characters of all languages. For more on Unicode, see Chapter 9, "I/O and Streams". Usually a `char` literal can be expressed by enclosing the desired character in single quotes, as shown here:

```
char c = 'w';
```

Of course, this technique works only if the desired character is available on the keyboard at hand. Another way to express a `char` literal is as a Unicode value specified using four hexadecimal digits, preceded by `\u`, with the entire expression in single quotes. For example:

```
char c1 = '\u4567';
```

Java supports a few escape sequences for denoting special characters:

- `'\n'` for new line
- `'\r'` for return
- `'\t'` for tab
- `'\b'` for backspace
- `'\f'` for formfeed
- `'\''` for single quote
- `'\" '` for double quote
- `'\\'` for backslash

*Integral literals* may be assigned to any numeric primitive data type. They may be expressed in decimal, octal, or hexadecimal. The default is decimal. To indicate octal, prefix the literal with `0` (zero). To indicate hexadecimal, prefix the literal with `0x` or `0X`; the hex digits may be upper- or lowercase. The value 28 may thus be expressed six ways:

- `28`
- `034`
- `0x1c`
- `0x1C`
- `0X1c`
- `0X1C`

By default, an integral literal is a 32-bit value. To indicate a `long` (64-bit) literal, append the suffix `L` to the literal expression. (The suffix can be lowercase, but then it looks so much like a one that your readers are bound to be confused.)

A *floating-point* literal expresses a floating-point number. In order to be interpreted as a floating-point literal, a numerical expression must contain one of the following:

- A decimal point, such as `1.414`
- The letter `E` or `e`, indicating scientific notation, such as `4.23E+21`
- The suffix `F` or `f`, indicating a `float` literal, such as `1.828f`
- The suffix `D` or `d`, indicating a `double` literal, such as `1234d`

A floating-point literal with no `F` or `D` suffix defaults to `double` type.

## String Literals

A *string literal* is a sequence of characters enclosed in double quotes. For example:

```
String s = "Characters in strings are 16-bit Unicode.";
```

Java provides many advanced facilities for specifying non-literal string values, including a concatenation operator and some sophisticated constructors for the `String` class. These facilities are discussed in detail in Chapter 8, "The `java.lang` and `java.util` Packages."

# Arrays

A Java *array* is an ordered collection of primitives, object references, or other arrays. Java arrays are homogeneous: except as allowed by polymorphism, all elements of an array must be of the same type. That is, when you create an array, you specify the element type, and the resulting array can contain only elements that are instances of that class or subclasses of that class.

To create and use an array, you must follow three steps:

1. Declaration
2. Construction
3. Initialization

Declaration tells the compiler the array's name and what type its elements will be. For example:

```
1. int[] ints;
2. Dimension[] dims;
3. float[][] twoDee;
```

Line 1 declares an array of a primitive type. Line 2 declares an array of object references (`Dimension` is a class in the `java.awt` package). Line 3 declares a two-dimensional array—that is, an array of arrays of `float`s.

The square brackets can come before or after the array variable name. This is also true, and perhaps most useful, in method declarations. A method that takes an array of `double`s could be declared as `myMethod(double dubs[])` or as `myMethod(double[] dubs)`; a method that returns an array of doubles may be declared as either `double[] anotherMethod()` or as `double anotherMethod()[]`. In this last case, the first form is probably more readable.

Generally, placing the square brackets adjacent to the type, rather than following the variable or method, allows the type declaration part to be read as a single unit: `int array` or `float array`, which might make more sense. However, C/C++ programmers will be more familiar with the form where the brackets are placed to the right of the variable or method declaration. Given the number of magazine articles that have been dedicated to ways to correctly interpret complex C/C++ declarations (perhaps you recall the "spiral rule"), it's probably not a bad thing that Java has modified the syntax for these declarations. Either way, you need to recognize both forms.

Notice that the declaration does not specify the size of an array. Size is specified at runtime, when the array is allocated via the `new` keyword. For example

```
1. int[] ints;          // Declaration to the compiler
2. ints = new int[25];  // Runtime construction
```

Since array size is not used until runtime, it is legal to specify size with a variable rather than a literal:

```
1. int size = 1152 * 900;
2. int[] raster;
3. raster = new int[size];
```

Declaration and construction may be performed in a single line:

```
1. int[] ints = new int[25];
```

When an array is constructed, its elements are automatically initialized to their default values. These defaults are the same as for object member variables. Numerical elements are initialized to 0; non-numeric elements are initialized to 0-like values, as shown in Table 1.5.

Arrays are actually objects, even to the extent that you can execute methods on them (mostly the methods of the `Object` class), although you cannot subclass the array class. So this initialization is exactly the same as for other objects, and as a consequence you will see this table again in the next section.

**TABLE 1.5** Array Element Initialization Values

| Element Type | Initial Value |
|---|---|
| byte | 0 |
| int | 0 |
| float | 0.0f |
| char | '\u0000' |
| object reference | null |
| short | 0 |
| long | 0L |
| double | 0.0d |
| boolean | false |

If you want to initialize an array to values other than those shown in Table 1.5, you can combine declaration, construction, and initialization into a single step. The following line of code creates a custom-initialized array of five floats:

```
1. float[] diameters = {1.1f, 2.2f, 3.3f, 4.4f, 5.5f};
```

The array size is inferred from the number of elements within the curly braces.

Of course, an array can also be initialized by explicitly assigning a value to each element, starting at array index 0:

```
1. long[] squares;
2. squares = new long[6000];
3. for (int i = 0; i < 6000; i++) {
4.   squares[i] = i * i;
5. }
```

When the array is created at line 2, it is full of default values (`0L`), which are replaced in lines 3–4. The code in the example works but can be improved. If you later need to change the array size (in line 2), the loop counter will have to change (in line 3), and the program could be damaged if line 3 is not taken care of. The safest way to refer to the size of an array is to append the `.length` member variable to the array name. Thus, our example becomes

```
1. long[] squares;
2. squares = new long[6000];
```

```
3. for (int i = 0; i < squares.length; i++) {
4.   squares[i] = i * i;
5. }
```

When an array has more than one dimension, there is more going on than you might think. Consider this declaration plus initialization:

```
int[][] myInts = new int[3][4];
```

It's natural to assume that the `myInts` contains 12 ints and to imagine them as organized into rows and columns, as shown in Figure 1.1.

Actually, Figure 1.1 is misleading. `myInts` is actually an array with three elements. Each element is a reference to an array containing 4 ints, as shown in Figure 1.2.

The subordinate arrays in a multi-dimension array don't have to all be the same length. It's possible to create an array that looks like Figure 1.3.

**FIGURE 1.1** The wrong way to think about multi-dimension arrays

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 91 | 92 | 93 | 94 |
| 2001 | 2002 | 2003 | 2004 |

**FIGURE 1.2** The right way to think about multi-dimension arrays

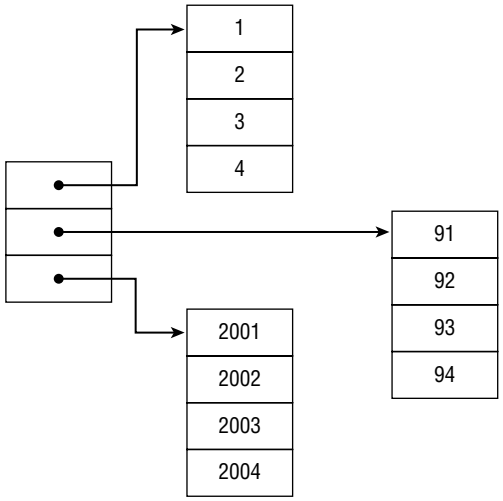


**FIGURE 1.3** An irregular multi-dimension array

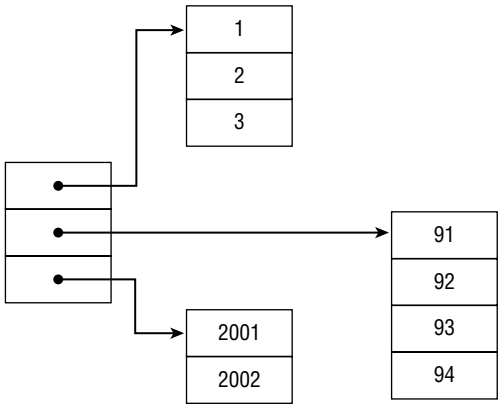



Figure 1.3 shows an array whose elements are an array of 3 ints, an array of 4 ints, and an array of 2 ints. Such an array may be created like this:

```
int[][] myInts = { {1, 2, 3}, {91, 92, 93, 94}, {2001, 2002} };
```

When you realize that the outermost array is a single-dimension array containing references, you understand that you can replace any of the references with a reference to a different subordinate array, provided the new subordinate array is of the right type. For example, you can do the following:

```
int[][] myInts = { {1, 2, 3}, {91, 92, 93, 94}, {2001, 2002} };
int[] replacement = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
myInts[1] = replacement;
```

# Importing

The term "import" can be confusing. In common speech, import means to bring something from abroad into one's own territory. In the Java context, it's natural to wonder what is getting brought in, and where it is getting brought into. A common mistake is to guess that importing has something to do with class loading. It's a reasonable mistake, since the class loader is the only obvious Java entity that brings something (class definitions) into somewhere (the Java Virtual Machine). However, the guess is dead wrong.

What gets brought in is the import class' name. The name is brought into the source file's *namespace*. A namespace is a kind of place—not a physical place, but an abstract place such as

a directory or a source file—that contains items with unique names. The easiest example is a directory: within a directory, all filenames must be different from all other filenames. Names may be duplicated in different namespaces. For example, `readme.txt` may appear only once within a single directory but may appear in any other directory.

Items that appear in namespaces have short names and long names. The short name is for use within the namespace. The long name is for use outside the namespace. Suppose directory `C:\MyCode\Projects` contains a file named `Sphinx.java`. When you are working in `C:\MyCode\Projects`, you can refer to the file by its short name: `Sphinx.java`. However, when your working directory is not `C:\MyCode\Projects`, you need to use the file's full name: `C:\MyCode\Projects\Sphinx.java`.

The namespace of a Java source file contains the names of all classes and interfaces in the source file's package. In other words, within the source file you may refer to any class by its short name; classes outside the package must be called by their complete names. Suppose the current package contains a class named `Formula`. The following code creates an instance of `Formula` and an instance of `Vector`:

```
1. Formula f = new Formula();
2. java.util.Vector vec = new java.util.Vector();
```

Line 2 is a mess. The `Vector` class resides in the `java.util` package, so it must be referred to by its full name…twice! (Once in the declaration, and again in the constructor call.) If there were no workaround, the only thing worse than writing Java code would be reading Java code. Fortunately, Java provides a workaround. The source file needs an `import` statement:

```
import java.util.Vector;
```

Then line 2 becomes

```
2. Vector vec = new Vector();
```

This statement imports the name "`Vector`" into the namespace, allowing it to be used without the "`java.util`" prefix. When the compiler encounters a short class name, it checks the current package. If the class name is not found, the compiler then checks its import statements. In our example, the compiler will notice that there is no `Vector` class in the current package, but there is an import statement. The `import` tells the compiler, "When I say `Vector`, I really mean `java.util.Vector`."

Java's `static import` facility, which was introduced in rev 5.0, allows you to import static data and methods, as well as classes. In other words, you may refer to static data and methods in external classes without using full names. For example, the `java.awt.Color` class contains static data members names `RED`, `GREEN`, `BLUE`, and so on. Suppose you want to set `myColor` to `GREEN`. Without static imports, you have to do the following:

```
import java.awt.Color;
…
myColor = Color.GREEN;
```

With a static import, you can import the name "GREEN" into your namespace:

```
import static java.awt.Color.GREEN;
…
myColor = GREEN;
```

Note that the `import` keyword is followed by `static`. This tells the compiler to import the name of a static element of a class, rather than a class name.

Static imports eliminate the nuisance of constant interfaces. Constant interfaces are fairly common, since before rev 5.0 there was no good alternative. Many packages or applications define constants that are needed by more than one source file. For example, an application that uses both English and metric weights might need the following:

```
public static float LBS_PER_KG = 2.2f;
public static float KGS_PER_LB = 1 / LBS_PER_KG;
```

Now the question is, where do these lines belong? The general answer to this question is that they belong in the most appropriate class or interface. Unfortunately, "most appropriate" doesn't always mean most convenient. Suppose you put our two lines in a class called `Scales`. Since the constants are in the namespace of `Scales`, they may appear there without prefix. For example, `Scales` might contain

```
massInPounds = massInKgs * LBS_PER_KG;
```

However, other classes must go to more trouble. Any class except `Scales` has to do the following:

```
massInPounds = massInKgs * Scales.LBS_PER_KG;
```

Many programmers, wishing to avoid the inconvenience of prefixing, have discovered the trick of creating an interface (known as a *constant interface*) to contain constants. This trick has two benefits. First, you don't have to decide which class to put the constants in; they go in the interface. Second, in any class that implements the constant interface, you don't have to prefix the constants. In our example, you might be tempted to place the constant definitions in an interface called `Conversion`. Then the `Scales` class, and all other classes that convert between pounds and kilos, can implement `Conversion`.

Unfortunately, constant interfaces have several drawbacks. In the first place, to say that a class implements an interface really means that the class exposes the public methods listed in the interface. Interfaces are for defining types and should be used exclusively for that purpose. Constant interfaces only contain data, so they definitely don't define types.

The second disadvantage is a bit more complicated. Suppose someone you work with writes some code that uses an instance of `Scales`. This person can legally reference that instance with a variable of type `Conversion`, even though doing so would be quite inappropriate. Later, if you wanted to eliminate the `Conversion` interface, you couldn't do so, because your misguided colleague would be relying on the existence of the interface.

With static imports, you have an alternative to constant interfaces. To use static imports, you first locate your constants in the classes where they belong. Let's assume you put `LBS_PER_KG` and `KGS_PER_LB` in the `Scales` class. Now any other source file can use the following syntax:

```
import static Scales.LBS_PER_KG;
import static Scales.KGS_PER_LB;
```

Any source file that uses these statements may refer to `LBS_PER_KG` and `KGS_PER_LB`, rather than `Scales.LBS_PER_KG` and `Scales.KGS_PER_LB`.

The `static import` facility is aware of packages and access modes. To do a static import from a class in a different package, you have to prefix the class name with its package path. For example, to import the constant `NORTH` from class `java.awt.BorderLayout`, you would use

```
import static java.awt.BorderLayout.NORTH;
```

Only public data may be imported from classes in external packages. Data imported from other classes in the same package may be public, protected, or default, but not private. These rules are consistent with the meanings of public, protected, default, and private.


Java's access modes are discussed in detail in Chapter 3, "Modifiers."

You can use the star notation to import all accessible constants from a class. The line

```
import static pkga.pkgb.AClassName.*;
```

will import all non-private constants if `AClassName` is in the current package or all public constants if `AClassName` is in a different package.

Static importing gives you access to static methods as well as static data. Suppose class `measure.Scales` has a method called `poundsToMicrograms()` that looks like this:

```
public static float poundsToMicrograms(float pounds) {
  return pounds * KGS_PER_LB * 1.0e6f;
}
```

Any source file can import this method as follows:

```
import static measure.Scales.poundsToMicrograms;
```

A source file that performs this import may invoke the method as (for example)

```
float ugs = poundsToMicrograms(lbs);
```

This is a bit more convenient than

```
float ugs = Scales.poundsToMicrograms(lbs);
```

As with ordinary imports, static imports have only a slight compile-time cost and zero runtime cost. Many programmers are unclear on this point, perhaps because the word "import" feels like such an active verb; it seems as if surely the class loader or some other mechanism must be hard at work. Remember that importing does nothing more than bring a name into the local namespace. So importing and static importing are quite inexpensive.

# Class Fundamentals

Java is all about classes, and a review of the exam objectives will show that you need to be intimately familiar with them. Classes are discussed in detail in Chapter 6, "Objects and Classes." For now, let's examine a few fundamentals.

## Class Paths

When the Java compiler or the Virtual Machine needs a classfile, it searches all the locations listed in its classpath. The classpath is formed by merging the `CLASSPATH` environment variable and any locations specified in `-classpath` or `-cp` command line arguments. The members of a classpath may be directories or jar files.

Let's take an example. Suppose the compiler is looking for class sgsware.sphinx.Domain. The package structure sgsware.sphinx requires that the Domain.class file must be in a directory called sphinx, which must be in a directory called sgsware. So the compiler checks each classpath member to see if it contains sgsware\sphinx\Domain.class.

On Windows platforms, directories and jar files in a classpath are separated by a semicolon (";"). On UNIX platforms the separator is a colon (":").

## The *main()* Method

The `main()` method is the entry point for standalone Java applications. To create an application, you write a class definition that includes a `main()` method. To execute an application, type **`java`** at the command line, followed by the name of the class containing the `main()` method to be executed.

The signature for `main()` is

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

The `main()` method must be public so that the JVM can call it. It is static so that it can be executed without the necessity of constructing an instance of the application class. The return type must be `void`.

The argument to `main()` is a single-dimension array of Strings, containing any arguments that the user might have entered on the command line. For example, consider the following command line:

```
% java Mapper France Belgium
```

With this command line, the `args[]` array has two elements: `France` in `args[0]`, and `Belgium` in `args[1]`. Note that neither the class name (`Mapper`) nor the command name (`java`) appears in the array. Of course, the name `args` is purely arbitrary: any legal identifier may be used, provided the array is a single-dimension array of `String` objects.

## Variables and Initialization

Java supports variables of three different lifetimes:

**Member variable** A *member variable* of a class is created when an instance is created, and it is destroyed when the object is destroyed. Subject to accessibility rules and the need for a reference to the object, member variables are accessible as long as the enclosing object exists.

**Automatic variable** An *automatic variable* of a method is created on entry to the method and exists only during execution of the method, and therefore it is accessible only during the execution of that method. (You'll see an exception to this rule when you look at inner classes, but don't worry about that for now.)

**Class variable** A *class variable* (also known as a *static variable*) is created when the class is loaded and is destroyed when the class is unloaded. There is only one copy of a class variable, and it exists regardless of the number of instances of the class, even if the class is never instantiated.

All member variables that are not explicitly assigned a value upon declaration are automatically assigned an initial value. The initialization value for member variables depends on the member variable's type. Values are listed in Table 1.6.

The values in Table 1.6 are the same as those in Table 1.5; member variable initialization values are the same as array element initialization values.

A member value may be initialized in its own declaration line:

```
1. class HasVariables {
2.   int x = 20;
3.   static int y = 30;
```

When this technique is used, nonstatic instance variables are initialized just before the class constructor is executed; here `x` would be set to 20 just before invocation of any `HasVariables` constructor. Static variables are initialized at class load time; here `y` would be set to 30 when the `HasVariables` class is loaded.

Automatic variables (also known as *method local variables* are not initialized by the system; every automatic variable must be explicitly initialized before being used. For example, this method will not compile:

```
1. public int wrong() {
2.   int i;
3.   return i+5;
4. }
```

The compiler error at line 3 is, "Variable `i` may not have been initialized." This error often appears when initialization of an automatic variable occurs at a lower level of curly braces than the use of that variable. For example, the following method returns the fourth root of a positive number:

```
1. public double fourthRoot(double d) {
2.   double result;
3.     if (d >= 0) {
4.       result = Math.sqrt(Math.sqrt(d));
5.     }
6.     return result;
7. }
```

Here the result is initialized on line 4, but the initialization takes place within the curly braces of lines 3 and 5. The compiler will flag line 6, complaining that "Variable `result` may not have been initialized." A common solution is to initialize `result` to some reasonable default as soon as it is declared:

```
1. public double fourthRoot(double d) {
2.   double result = 0.0;  // Initialize
3.     if (d >= 0) {
4.       result = Math.sqrt(Math.sqrt(d));
5.     }
6.     return result;
7. }
```

Now `result` is satisfactorily initialized. Line 2 demonstrates that an automatic variable may be initialized in its declaration line. Initialization on a separate line is also possible.

Class variables are initialized in the same manner as for member variables.

**TABLE 1.6** Initialization Values for Member Variables

| Element Type | Initial Value | Element Type | Initial Value |
|---|---|---|---|
| `byte` | 0 | `short` | 0 |
| `int` | 0 | `long` | 0L |
| `float` | 0.0f | `double` | 0.0d |
| `char` | '\u0000' | `boolean` | false |
| `object reference` | `null` | | |

# Argument Passing: By Reference or by Value

When Java passes an argument into a method call, a *copy* of the argument is actually passed. Consider the following code fragment:

```
1. double radians = 1.2345;
2. System.out.println("Sine of " + radians +
3.                    " = " + Math.sin(radians));
```

The variable `radians` contains a pattern of bits that represents the number 1.2345. On line 2, a copy of this bit pattern is passed into the method-calling apparatus of the JVM.

When an argument is passed into a method, changes to the argument value by the method do not affect the original data. Consider the following method:

```
1. public void bumper(int bumpMe) {
2.   bumpMe += 15;
3. }
```

Line 2 modifies a copy of the parameter passed by the caller. For example

```
1. int xx = 12345;
2. bumper(xx);
3. System.out.println("Now xx is " + xx);
```

On line 2, the caller's `xx` variable is copied; the copy is passed into the `bumper()` method and incremented by 15. Because the original `xx` is untouched, line 3 will report that `xx` is still 12345.

This is also true when the argument to be passed is an object rather than a primitive. However, it is crucial for you to understand that the effect is very different. In order to understand the process, you have to understand the concept of the *object reference*.

Java programs do not deal directly with objects. When an object is constructed, the constructor returns a value—a bit pattern—that uniquely identifies the object. This value is known as a *reference* to the object. For example, consider the following code:

```
1. Button btn;
2. btn = new Button("Ok");
```

In line 2, the `Button` constructor returns a reference to the just-constructed button—not the actual button object or a copy of the button object. This reference is stored in the variable `btn`. In some implementations of the JVM, a reference is simply the address of the object; however, the JVM specification gives wide latitude as to how references can be implemented. You can think of a reference as simply a pattern of bits that uniquely identifies an individual object.

### How to Create a Reference to a Primitive

This is a useful technique if you need to create the effect of passing primitive values by reference. Simply pass an array of one primitive element over the method call, and the called method can now change the value seen by the caller. To do so, use code like this:

```
1. public class PrimitiveReference {
2.   public static void main(String args[]) {
3.     int [] myValue = { 1 };
4.     modifyIt(myValue);
5.     System.out.println("myValue contains " +
6.                        myValue[0]);
7.   }
8.   public static void modifyIt(int [] value) {
9.     value[0]++;
10.  }
11. }
```

In most JVMs, the reference value is actually the address of an address. This second address refers to the real data. This approach, called *double indirection*, allows the garbage collector to relocate objects to reduce memory fragmentation.

When Java code appears to store objects in variables or pass objects into method calls, the object references are stored or passed.

Consider this code fragment:

```
1. Button btn;
2. btn = new Button("Pink");
3. replacer(btn);
4. System.out.println(btn.getLabel());
5.
6. public void replacer(Button replaceMe) {
7.   replaceMe = new Button("Blue");
8. }
```

Line 2 constructs a button and stores a reference to that button in `btn`. In line 3, a copy of the reference is passed into the `replacer()` method. Before execution of line 7, the value in `replaceMe` is a reference to the Pink button. Then line 7 constructs a second button and stores a reference to the second button in `replaceMe`, thus overwriting the reference to the Pink button.

However, the caller's copy of the reference is not affected, so on line 4 the call to `btn.getLabel()` calls the original button; the string printed out is "Pink".

You have seen that called methods cannot affect the original value of their arguments—that is, the values stored by the caller. However, when the called method operates on an object via the reference value that is passed to it, there are important consequences. If the method modifies the object via the reference, as distinguished from modifying the method argument—the reference—then the changes will be visible to the caller. For example

```
1. Button btn;
2. btn = new Button("Pink");
3. changer(btn);
4. System.out.println(btn.getLabel());
5.
6. public void changer(Button changeMe) {
7.   changeMe.setLabel("Blue");
8. }
```

In this example, the variable `changeMe` is a copy of the reference `btn`, just as before. However, this time the code uses the copy of the reference to change the actual original object rather than trying to change the reference. Because the caller's object is changed rather than the callee's reference, the change is visible and the value printed out by line 4 is "Blue".

Arrays are objects, meaning that programs deal with references to arrays, not with arrays themselves. What gets passed into a method is a copy of a reference to an array. It is therefore possible for a called method to modify the contents of a caller's array.

# Garbage Collection

Most modern languages permit you to allocate data storage during a program run. In Java, this is done directly when you create an object with the `new` operation and indirectly when you call a method that has local variables or arguments. Method locals and arguments are allocated space on the stack and are discarded when the method exits, but objects are allocated space on the heap and have a longer lifetime.


Each process has its own stack and heap, and they are located on opposite sides of the process address space. The sizes of the stack and heap are limited by the amount of memory that is available on the host running the program. They may be further limited by the operating system or user-specific limits.

It is important to recognize that objects are always allocated on the heap. Even if they are created in a method using code like

```
public void aMethod() {
   MyClass mc = new MyClass();
}
```

the local variable `mc` is a reference, allocated on the stack, whereas the object to which that variable refers, an instance of `MyClass`, is allocated on the heap.

In this discussion, we are concerned with recovery of space allocated on the heap. The increased lifetime raises the question of when storage allocation on the heap can be released. Some languages require that you, the programmer, explicitly release the storage when you have finished with it. This approach has proven seriously error-prone, because you might release the storage too soon (causing corrupted data if any other reference to the data is still in use) or forget to release it altogether (causing a memory shortage). Java's garbage collection solves the first of these problems and greatly simplifies the second.

### How to Cause Leaks in a Garbage Collection System

The nature of automatic garbage collection has an important consequence: you can still get memory leaks. If you allow live, accessible references to unneeded objects to persist in your programs, then those objects cannot be garbage collected. Therefore, it may be a good idea to explicitly assign `null` into a variable when you have finished with it. This issue is particularly noticeable if you are implementing a collection of some kind.

In this example, assume the array `storage` is being used to maintain the storage of a stack. This `pop()` method is inappropriate:

```
1. public Object pop() {
2.   return storage[index--];
3. }
```

If the caller of this `pop()` method abandons the popped value, it will not be eligible for garbage collection until the array element containing a reference to it is overwritten. This might take a long time. You can speed up the process like this:

```
1. public Object pop() {
2.   Object returnValue = storage[index];
3.   storage[index--] = null;
4.   return returnValue;
5. }
```

In Java, you never explicitly free memory that you have allocated; instead, Java provides automatic garbage collection. The runtime system keeps track of the memory that is allocated and is able to determine whether that memory is still useable. This work is usually done in the background by a low-priority thread that is referred to as the *garbage collector*. When the garbage collector finds memory that is no longer accessible from any live thread (the object is out of scope), it takes steps to release it back into the heap for re-use. Specifically, the garbage collector calls the class destructor method called `finalize()` (if it is defined) and then frees the memory.

Garbage collection can be done in a number of different ways; each has advantages and disadvantages, depending on the type of program that is running. A real-time control system, for example, needs to know that nothing will prevent it from responding quickly to interrupts; this application requires a garbage collector that can work in small chunks or that can be interrupted easily. On the other hand, a memory-intensive program might work better with a garbage collector that stops the program from time to time but recovers memory more urgently as a result. At present, garbage collection is hardwired into the Java runtime system; most garbage collection algorithms use an approach that gives a reasonable compromise between speed of memory recovery and responsiveness. In the future, you will probably be able to plug in different garbage-collection algorithms or buy different JVMs with appropriate collection algorithms, according to your particular needs.

This discussion leaves one crucial question unanswered: When is storage recovered? The best answer is that storage is not recovered unless it is definitely no longer in use. That's it. Even though you are not using an object any longer, you cannot say if it will be collected in 1 millisecond, in 100 milliseconds, or even if it will be collected at all. The methods `System.gc()` and `Runtime.gc()` look as if they run the garbage collector, but even these cannot be relied upon in general, because some other thread might prevent the garbage-collection thread from running. In fact, the documentation for the `gc()` methods states:

> Calling this method suggests that the Java Virtual Machine expends effort toward recycling unused objects.

# Summary

This chapter has covered a variety of topics. You learned that a source file's elements must appear in this order:

1. Package declaration
2. Import statements
3. Class, interface, and enum definitions

Imports may be static. There should be, at most, one public class definition per source file; the filename must match the name of the public class.

You also learned that an identifier must begin with a letter, a dollar sign, or an underscore; subsequent characters may be letters, dollar signs, underscores, or digits. Java has four signed

integral primitive data types: `byte`, `short`, `int`, and `long`; all four types display the behavior of two's-complement representation. Java's two floating-point primitive data types are `float` and `double`; the `char` type is unsigned and represents a Unicode character; the `boolean` type may take on only the values `true` and `false`.

In addition, you learned that arrays must be (in order)

1. Declared
2. Constructed
3. Initialized

Default initialization is applied to member variables, class variables, and array elements, but not automatic variables. The default values are `0` for numeric types, the `null` value for object references, the `null` character for `char`, and `false` for `boolean`. The `length` member of an array gives the number of elements in the array. A class with a `main()` method can be invoked from the command line as a Java application. The signature for `main()` is `public static void main(String[] args)`. The `args[]` array contains all command-line arguments that appeared after the name of the application class.

You should also understand that method arguments are copies, not originals. For arguments of primitive data type, this means that modifications to an argument within a method are not visible to the caller of the method. For arguments of object type (including arrays), modifications to an argument value within a method are still not visible to the caller of the method; however, modifications in the object or array to which the argument refers *do* appear to the caller.

Finally, Java's garbage-collection mechanism may recover only memory that is definitely unused. It is not possible to force garbage collection reliably. It is not possible to predict when a piece of unused memory will be collected, only to say when it becomes *eligible* for collection. Garbage collection does not prevent memory leaks; they can still occur if unused references are not cleared to `null` or destroyed.

# Exam Essentials

**Recognize and create correctly constructed source files.** You should know the various kinds of compilation units and their required order of appearance.

**Recognize and create correctly constructed declarations.** You should be familiar with declarations of packages, classes, interfaces, methods, and variables.

**Recognize Java keywords.** You should recognize the keywords and reserved words listed in Table 1.1.

**Distinguish between legal and illegal identifiers.** You should know the rules that restrict the first character and the subsequent characters of an identifier.

**Know all the primitive data types and the ranges of the integral data types.** These are summarized in Tables 1.2 and 1.3.

**Recognize correctly formatted literals.** You should be familiar with all formats for literal characters, strings, and numbers.

**Know how to declare and construct arrays.** The declaration includes one empty pair of square brackets for each dimension of the array. The square brackets can appear before or after the array name. Arrays are constructed with the keyword `new`.

**Know the default initialization values for all possible types of class variables and array elements. Know when data is initialized.** Initialization takes place when a class or array is constructed. The initialization values are `0` for numeric type arrays, `false` for `boolean` arrays, and `null` for object reference type arrays.

**Understand importing and static importing.** Be aware of the difference between traditional importing and the new `static import` facility.

**Know the contents of the argument list of an application's `main()` method, given the command line that invoked the application.** Be aware that the list is an array of Strings containing everything on the command line except the `java` command, command-line options, and the name of the class.

**Know that Java passes method arguments by value.** Changes made to a method argument are not visible to the caller, because the method argument changes a copy of the argument. Objects are not passed to methods; only references to objects are passed.

**Understand memory reclamation and the circumstances under which memory will be reclaimed.** If an object is still accessible to any live thread, that object will certainly not be collected. This is true even if the program will never access the object again—the logic is simple and cannot make inferences about the semantics of the code. No guarantees are made about reclaiming available memory or the timing of reclamation if it does occur. A standard JVM has no entirely reliable, platform-independent way to force garbage collection. The `System` and `Runtime` classes each have a `gc()` method, and these methods make it more likely that garbage collection will run, but they provide no guarantee.

# Review Questions

1. A signed data type has an equal number of non-zero positive and negative values available.
   - **A.** True
   - **B.** False

2. Choose the valid identifiers from those listed here. (Choose all that apply.)
   - **A.** `BigOlLongStringWithMeaninglessName`
   - **B.** `$int`
   - **C.** `bytes`
   - **D.** `$1`
   - **E.** `finalist`

3. Which of the following signatures are valid for the `main()` method entry point of an application? (Choose all that apply.)
   - **A.** `public static void main()`
   - **B.** `public static void main(String arg[])`
   - **C.** `public void main(String [] arg)`
   - **D.** `public static void main(String[] args)`
   - **E.** `public static int main(String [] arg)`

4. If all three top-level elements occur in a source file, they must appear in which order?
   - **A.** Imports, package declarations, classes/interfaces/enums
   - **B.** Classes/interfaces/enums, imports, package declarations
   - **C.** Package declaration must come first; order for imports and class/interfaces/enum definitions is not significant
   - **D.** Package declaration, imports, class/interface/enum definitions.
   - **E.** Imports must come first; order for package declarations and class/interface/enum definitions is not significant

5. Consider the following line of code:

```
int[] x = new int[25];
```

   After execution, which statements are true? (Choose all that apply.)
   - **A.** `x[24]` is 0
   - **B.** `x[24]` is undefined
   - **C.** `x[25]` is 0
   - **D.** `x[0]` is `null`
   - **E.** `x.length` is 25

**6.** Consider the following application:

```
1. class Q6 {
2.   public static void main(String args[]) {
3.     Holder h = new Holder();
4.     h.held = 100;
5.     h.bump(h);
6.     System.out.println(h.held);
7.   }
8. }
9.
10. class Holder {
11.   public int held;
12.   public void bump(Holder theHolder) {
13.     theHolder.held++; }
14.   }
15. }
```

What value is printed out at line 6?

**A.** 0

**B.** 1

**C.** 100

**D.** 101

**7.** Consider the following application:

```
1. class Q7 {
2.   public static void main(String args[]) {
3.     double d = 12.3;
4.     Decrementer dec = new Decrementer();
5.     dec.decrement(d);
6.     System.out.println(d);
7.   }
8. }
9.
10. class Decrementer {
11.   public void decrement(double decMe) {
12.     decMe = decMe - 1.0;
13.  }
14. }
```

What value is printed out at line 6?

**A.** 0.0

**B.** 1.0

**C.** 12.3

**D.** 11.3

**8.** How can you force garbage collection of an object?

**A.** Garbage collection cannot be forced.

**B.** Call `System.gc()`.

**C.** Call `System.gc()`, passing in a reference to the object to be garbage-collected.

**D.** Call `Runtime.gc()`.

**E.** Set all references to the object to new values (`null`, for example).

**9.** What is the range of values that can be assigned to a variable of type `short`?

**A.** Depends on the underlying hardware

**B.** 0 through $2^{16} - 1$

**C.** 0 through $2^{32} - 1$

**D.** $-2^{15}$ through $2^{15} - 1$

**E.** $-2^{31}$ through $2^{31} - 1$

**10.** What is the range of values that can be assigned to a variable of type `byte`?

**A.** Depends on the underlying hardware

**B.** 0 through $2^{8} - 1$

**C.** 0 through $2^{16} - 1$

**D.** $-2^{7}$ through $2^{7} - 1$

**E.** $-2^{15}$ through $2^{15} - 1$

**11.** Suppose a source file contains a large number of `import` statements. How do the imports affect the time required to compile the source file?

**A.** Compilation takes no additional time.

**B.** Compilation takes slightly more time.

**C.** Compilation takes significantly more time.

**12.** Suppose a source file contains a large number of `import` statements and one class definition. How do the imports affect the time required to load the class?

**A.** Class loading takes no additional time.

**B.** Class loading takes slightly more time.

**C.** Class loading takes significantly more time.

**13.** Which of the following are legal `import` statements?

**A.** import java.util.Vector;

**B.** static import java.util.Vector.*;

**C.** import static java.util.Vector.*;

**D.** import java.util.Vector static;

**14.** Which of the following may be statically imported? (Choose all that apply.)

**A.** Package names

**B.** Static method names

**C.** Static field names

**D.** Method-local variable names

**15.** What happens when you try to compile and run the following code?

```
public class Q15 {
  static String s;
  public static void main(String[] args) {
    System.out.println(">>" + s + "<<");
  }
}
```

**A.** The code does not compile

**B.** The code compiles, and prints out >><<

**C.** The code compiles, and prints out >>null<<

**16.** Which of the following are legal? (Choose all that apply.)

**A.** `int a = abcd;`

**B.** `int b = ABCD;`

**C.** `int c = 0xabcd;`

**D.** `int d = 0XABCD;`

**E.** `int e = 0abcd;`

**F.** `int f = 0ABCD;`

**17.** Which of the following are legal? (Choose all that apply.)

**A.** `double d = 1.2d;`

**B.** `double d = 1.2D;`

**C.** `double d = 1.2d5;`

**D.** `double d = 1.2D5;`

**18.** Which of the following are legal?

**A.** `char c = 0x1234;`

**B.** `char c = \u1234;`

**C.** `char c = '\u1234';`

**19.** Consider the following code:

```
1. StringBuffer sbuf = new StringBuffer();
2. sbuf = null;
3. System.gc();
```

Choose all true statements:

**A.** After line 2 executes, the `StringBuffer` object is garbage collected.

**B.** After line 3 executes, the `StringBuffer` object is garbage collected.

**C.** After line 2 executes, the `StringBuffer` object is eligible for garbage collection.

**D.** After line 3 executes, the `StringBuffer` object is eligible for garbage collection.

**20.** Which of the following are true? (Choose all that apply.)

**A.** Primitives are passed by reference.

**B.** Primitives are passed by value.

**C.** References are passed by reference.

**D.** References are passed by value.

# Answers to Review Questions

1. B. The range of negative numbers is greater by one than the range of positive numbers.

2. A, B, C, D, E. All of the identifiers are valid. An identifier begins with a letter, a dollar sign, or an underscore; subsequent characters may be letters, dollar signs, underscores, or digits. And of course keywords and their kin may not be identifiers.

3. B, D. All the choices are valid method signatures. However, in order to be the entry point of an application, a `main()` method must be public, static, and void; it must take a single argument of type `String[]`.

4. D. Package declaration must come first, followed by imports, followed by class/interface/enum definitions.

5. A, E. The array has 25 elements, indexed from 0 through 24. All elements are initialized to 0.

6. D. A holder is constructed on line 3. A reference to that holder is passed into method `bump()` on line 5. Within the method call, the holder's `held` variable is bumped from 100 to 101.

7. C. The `decrement()` method is passed a copy of the argument `d`; the copy gets decremented, but the original is untouched.

8. A. Garbage collection cannot be forced. Calling `System.gc()` or `Runtime.gc()` is not 100 percent reliable, because the garbage-collection thread might defer to a thread of higher priority; thus options B and D are incorrect. Option C is incorrect because the two `gc()` methods do not take arguments; in fact, if you still have a reference to pass into any method, the object is not yet eligible to be collected. Option E will make the object eligible for collection the next time the garbage collector runs.

9. D. The range for a 16-bit `short` is $-2^{15}$ through $2^{15} - 1$. This range is part of the Java specification, regardless of the underlying hardware.

10. D. The range for an 8-bit `byte` is $-2^7$ through $2^7-1$. Table 1.3 lists the ranges for Java's integral primitive data types.

11. B . Importing slightly increases compilation time.

12. A.. Importing is strictly a compile-time function. It has no effect on class loading or on any other run-time function.

13. A, C. The `import` keyword may optionally be followed by the `static` keyword.

14. B, C. You may statically import method and field names.

15. C. The code compiles without error. At static initialization time, `s` is initialized to null (and not to a reference to an empty string, as suggested by C).

16. C, D. The characters a–f and A–F may be combined with the digits 0–9 to create a hexadecimal literal, which must begin with `0x`.

**17.** A, B. The d suffix in option A and the D suffix in option B are optional. Options C and D are illegal because the notation requires e or E, not d or D.

**18.** A, C. A legally assigns a literal numeric value to a char. To assign a literal unicode value, the literal must be enclosed in single quotes as in C.

**19.** C.. After line 2 executes, there are no references to the `StringBuffer` object, so it becomes eligible for garbage collection.

**20.** B, D. In Java, all arguments are passed by value.