Part I

The Programmer's Exam

1 Language Fundamentals
2 Declarations and Access Control
3 Operators and Assignments
4 Flow Control, Exceptions, and Assertions
5 Object Orientation, Overloading and Overriding, Constructors, and Return Types
6 Java.lang—The Math Class, Strings, and Wrappers
7 Objects and Collections
8 Inner Classes
9 Threads

1 Language Fundamentals

CERTIFICATION OBJECTIVES

- Java Programming Language Keywords
- Literals and Ranges of All Primitive Data Types
- Array Declaration, Construction, and Initialization
- Using a Variable or Array Element That Is Uninitialized and Unassigned
- Command-Line Arguments to Main

✓ Two-Minute Drill

Q&A Self Test

This chapter looks at the Java fundamentals that you need to pass the Java 1.4 Programmer exam. Because you're planning on becoming Sun certified, we assume you already know the basics of Java, so this chapter concentrates just on the details you'll need for the exam. If you're completely new to Java, this chapter (and the rest of the book) will be confusing, despite our spectacularly cogent writing. That's our story and we're sticking to it!

CERTIFICATION OBJECTIVE

Java Programming Language Keywords (Exam Objective 4.4)

Identify all Java programming language keywords and correctly constructed identifiers.

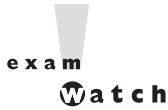
Keywords are special reserved words in Java that you cannot use as identifiers (names) for classes, methods, or variables. They have meaning to the compiler; it uses them to figure out what your source code is trying to do. Table 1-1 contains all 49 of the reserved keywords.

You *must* memorize these for the test; you can count on being asked to select the keywords (and nonkeywords) from a list. Notice none of the reserved words have

TABLE 1-1 Complete List of Java Keywords

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

capital letters; this is a good first step when weeding out nonkeywords on the exam. You're probably familiar with most of them, but we'll review them anyway. Don't worry right now about what each keyword means or does; we'll cover most of them in more detail in later chapters.



Look for questions that include reserved words from languages other than Java. You might see `include`, `overload`, `unsigned`, `virtual`, `friend`, and the like. Besides appearing in questions specifically asking for keyword identification, the "imposter" words may show up in code examples used anywhere in the exam. Repeat after me, "Java is not C++."

Access Modifiers

The following are access modifiers:

- **`private`** Makes a method or a variable accessible only from within its own class.
- **`protected`** Makes a method or a variable accessible only to classes in the same package or subclasses of the class.
- **`public`** Makes a class, method, or variable accessible from any other class.

Class, Method, and Variable Modifiers

The following are class, method, and/or variable modifiers:

- **`abstract`** Used to declare a class that cannot be instantiated, or a method that must be implemented by a nonabstract subclass.
- **`class`** Keyword used to specify a class.
- **`extends`** Used to indicate the superclass that a subclass is extending.
- **`final`** Makes it impossible to extend a class, override a method, or reinitialize a variable.
- **`implements`** Used to indicate the interfaces that a class will implement.
- **`interface`** Keyword used to specify an interface.
- **`native`** Indicates a method is written in a platform-dependent language, such as C.
- **`new`** Used to instantiate an object by invoking the constructor.

- **`static`** Makes a method or a variable belong to a class as opposed to an instance.
- **`strictfp`** Used in front of a method or class to indicate that floating-point numbers will follow FP-strict rules in all expressions.
- **`synchronized`** Indicates that a method can be accessed by only one thread at a time.
- **`transient`** Prevents fields from ever being serialized. *Transient* fields are always skipped when objects are serialized.
- **`volatile`** Indicates a variable may change out of sync because it is used in threads.

Flow Control

The following are keywords used to control the flow through a block of code:

- **`break`** Exits from the block of code in which it resides.
- **`case`** Executes a block of code, dependent on what the *switch* tests for.
- **`continue`** Stops the rest of the code following this statement from executing in a loop and then begins the next iteration of the loop.
- **`default`** Executes this block of code if none of the switch-case statements match.
- **`do`** Executes a block of code one time, then, in conjunction with the *while* statement, it performs a test to determine whether the block should be executed again.
- **`else`** Executes an alternate block of code if an *if* test is false.
- **`for`** Used to perform a conditional loop for a block of code.
- **`if`** Used to perform a logical test for *true* or *false*.
- **`instanceof`** Determines whether an object is an instance of a class, superclass, or interface.
- **`return`** Returns from a method without executing any code that follows the statement (can optionally return a variable).

- **`switch`** Indicates the variable to be compared with the *case* statements.
- **`while`** Executes a block of code repeatedly while a certain condition is *true*.

Error Handling

The following are keywords used in error handling:

- **`catch`** Declares the block of code used to handle an exception.
- **`finally`** Block of code, usually following a *try-catch* statement, which is executed no matter what program flow occurs when dealing with an exception.
- **`throw`** Used to pass an exception up to the method that called this method.
- **`throws`** Indicates the method will pass an exception to the method that called it.
- **`try`** Block of code that will be tried, but which may cause an exception.
- **`assert`** Evaluates a conditional expression to verify the programmer's assumption.

Package *Control*

The following are keywords used for package control:

- **`import`** Statement to import packages or classes into code.
- **`package`** Specifies to which package all classes in a source file belong.

Primitives

The following keywords are primitives:

- **`boolean`** A value indicating *true* or *false*.
- **`byte`** An 8-bit integer (signed).
- **`char`** A single Unicode character (16-bit unsigned)
- **`double`** A 64-bit floating-point number (signed).

- **`float`** A 32-bit floating-point number (signed).
- **`int`** A 32-bit integer (signed).
- **`long`** A 64-bit integer (signed).
- **`short`** A 16-bit integer (signed).

Variable Keywords

The following keywords are a special type of reference variable:

- **`super`** Reference variable referring to the immediate superclass.
- **`this`** Reference variable referring to the current instance of an object.

Void Return Type Keyword

The `void` keyword is used only in the return value placeholder of a method declaration.

- **`void`** Indicates no return type for a method.

Unused Reserved Words

There are two keywords that are reserved in Java but which are not used. If you try to use one of these, the Java compiler will scold you with the following:

```
KeywordTest.java:4: 'goto' not supported.
               goto MyLabel;
1 error
```

The engineers' first-draft of the preceding compiler warning resembled the following:

```
KeywordTest.java:4: ‘goto’ not supported. Duh.
You have no business programming in Java. Begin erasing Java
Software Development Kit? (Yes/OK)
1 life-altering error
```

- **`const`** Do not use to declare a constant; use `public static final`.
- **`goto`** Not implemented in the Java language. It's considered harmful.

Look for questions that use a keyword as the name of a method or variable. The question might* appear *to be asking about, say, a runtime logic problem, but the* real *problem will be that the code won't even compile because of the illegal use of a keyword. For example, the following code will not compile:

```
class Foo {
public void go() {
     // complex code here
    }
    public int break(int b) {
      // code that appears to break something
  }
}
```

You might be fooled by the use of the keyword `break` as a method name, because the method might genuinely appear to be code that "breaks" something, and therefore the method name makes sense. Meanwhile, you're trying to figure out the complex code within the methods, when you needn't look beyond the illegal method name and choose the "Code does not compile" answer.

According to the Java Language Specification, `null`, `true`, and `false` are technically literal values (sometimes referred to as manifest constants) and not keywords. Just as with the other keywords, if you try to create an identifier with one of these literal values, you'll get a compiler error. For the purposes of the exam, treat them just as you would the other reserved words. You will *not* be asked to differentiate between reserved words and these reserved literals.

Be careful of practice exams with questions that, for example, ask if* `false` *is a keyword. Many exam candidates worry about how to answer such a question, but the real exam does not expect you to make a distinction between the reserved keywords and the literals of* `null`*,* `true`*, and* `false`*. Because the certainty of this being on the exam has reached urban legend status, Sun modified the objectives for exam 310-035 to clear up any confusion. Objective 4.4 now includes the statement, "Note: There will not be any questions regarding esoteric distinctions between keywords and manifest constants." Contrary to popular belief, the exam creators are not evil or malicious. (I will admit, however, that while creating the exam, we experienced a giddy joy when one of us came up with a particularly tricky, er, clever question. High-fives all around!)

```
class LiteralTest {
     public static void main (String [] args) {
          int true = 100; // this will cause error
     }
}
```

Compiling this code gives us the following error (or something similar depending on which compiler you are using):

```
%javac LiteralTest.java
LiteralTest.java:3: not a statement.
                  int true = 100; // this will cause error
                  ^
```

In other words, trying to assign a value to *true* is much like saying:

```
int 200 = 100;
```

exam Watch

***Look for words that differ from the Java reserved words in subtle ways. For example, you might see* protect *rather than* `protected`, extend *rather than* `extends`.**

CERTIFICATION OBJECTIVE

Literals and Ranges of All Primitive Data Types (Exam Objective 4.6)

State the range of all primitive data types and declare literal values for String and all primitive types using all permitted formats, bases, and representations.

For the exam, you'll need to know the ranges of all primitive data types. Primitives include `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. The primitive `long`, for instance, has a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. But you knew that. Go memorize them all and come back when you've burned it in. *Just kidding.* The good news is you don't have to memorize such ridiculous numbers. There's an easier method to calculate the ranges, and for the larger integer values it will be enough to know that 16 bits gives you

more than 60,000 possibilities, 32 bits gives you approximately 4 billion, and so on. But you *will* need to know that the number types (both integer and floating-point types) are all signed, and how that affects the range. First, let's review the concepts.

Range of Primitive Types

All six number types in Java are signed, meaning they can be negative or positive. The leftmost bit (the most significant digit) is used to represent the sign, where a `1` means negative (glass half empty) and `0` means positive (glass half full), as shown in Figure 1-1. The rest of the bits represent the value, using two's complement notation.

Table 1-2 shows the primitive types with their sizes and ranges. Figure 1-2 shows that with a byte, for example, there are 256 possible numbers (or 2^8). Half of these are negative, and half -1 are positive. The positive range is one less than the negative range because the number zero is stored as a positive binary number. We use the formula $-2^{(bits - 1)}$ to calculate the negative range, and we use $2^{(bits - 1)}–1$ for the positive range.

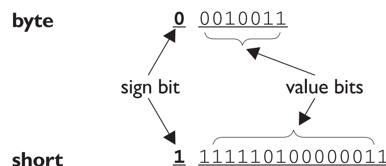
The range for floating-point numbers is complicated to determine, but luckily you don't need to know these for the exam (although you *are* expected to know that a `double` holds 64 bits and a `float` 32).

For `boolean` types there is not a range; a `boolean` can be only `true` or `false`. If someone asks you for the bit depth of a boolean, look them straight in the eye and say, "That's virtual-machine dependent." They'll be impressed.

The `char` type (a character) contains a single, 16-bit Unicode character. Although the extended ASCII set known as ISO Latin-1 needs only 8 bits (256 different characters), a larger range is needed to represent characters found in languages other than English. Unicode characters are actually represented by unsigned 16-bit integers, which means 2^{16} possible values, ranging from 0 to 65535 (2^{16})-1. You'll learn in

FIGURE 1-1

The sign bit for a byte



sign bit: 0 = positive
1 = negative

value bits:

byte: 7 bits can represent 2^7 or 128 different values:
0 thru 127 -or- –128 thru –1

short: 15 bits can represent 2^{15} or 32768 values:
0 thru 127 -or- –32768 thru –1

TABLE 1-2 Ranges of Primitive Numbers

Type	Bits	Bytes	Minimum Range	Maximum Range
`byte`	8	1	-2^{7}	$2^{7}-1$
`short`	16	2	-2^{15}	$2^{15}-1$
`int`	32	4	-2^{31}	$2^{31}-1$
`long`	64	8	-2^{63}	$2^{63}-1$
`float`	32	4	Not needed	Not needed
`double`	64	8	Not needed	Not needed

Chapter 3 that because a `char` is really an integer type, it can be assigned to any number type large enough to hold 65535.

Literal Values for All Primitive Types

A primitive literal is merely a source code representation of the primitive data types—in other words, an integer, floating-point number, boolean, or character that you type in while writing code. The following are examples of primitive literals:

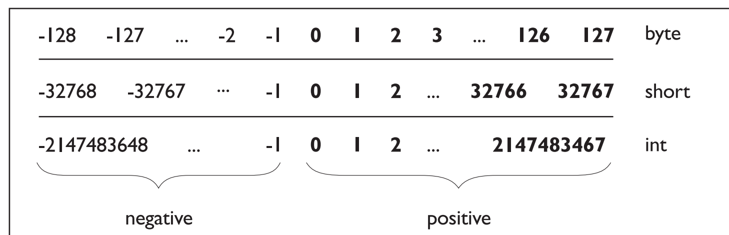
```
'b'  //  char literal
42  // int literal
false  // boolean literal
2546789.343  // double literal
```

Integer Literals

There are three ways to represent integer numbers in the Java language: decimal (base 10), octal (base 8), and hexadecimal (base 16). Most exam questions with integer literals use decimal representations, but the few that use octal or hexadecimal are worth studying for. Even though the odds that you'll ever actually *use* octal in the real world are astronomically tiny, they were included in the exam just for fun.

FIGURE 1-2

The range of a byte



Decimal Literals Decimal integers need no explanation; you've been using them since grade one or earlier. Chances are, you don't keep your checkbook in hex. (If you *do*, there's a Geeks Anonymous (GA) group ready to help.) In the Java language, they are represented as is, with no prefix of any kind, as follows:

```
int length = 343;
```

Octal Literals Octal integers use only the digits 0 to 7. In Java, you represent an integer in octal form by placing a zero in front of the number, as follows:

```
class Octal {
     public static void main(String [] args) {
          int five = 06; // Equal to decimal 6
          int seven = 07; // Equal to decimal 7
          int eight = 010; // Equal to decimal 8
          int nine = 011; // Equal to decimal 9
          System.out.println("Octal 010 = " + eight);
     }
}
```

Notice that when we get past seven and are out of digits to use (we are only allowed the digits 0 through 7 for octal numbers), we revert back to zero, and one is added to the beginning of the number. You can have up to 21 digits in an octal number, not including the leading zero. If we run the preceding program, it displays the following:

```
Octal 010 = 8
```

Hexadecimal Literals Hexadecimal (*hex* for short) numbers are constructed using 16 distinct symbols. Because we never invented single digit symbols for the numbers 10 through 15, we use alphabetic characters to represent these digits. Counting from 0 through 15 in hex looks like this:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

Java will accept capital or lowercase letters for the extra digits (one of the few places Java is not case-sensitive!). You are allowed up to 16 digits in a hexadecimal number, not including the prefix *0x* or the optional suffix extension *L*, which will be explained later.

All of the following hexadecimal assignments are legal:

```
class HexTest {
     public static void main (String [] args) {
          int x = 0X0001;
```

```
            int y = 0x7fffffff;
            int z = 0xDeadCafe;
            System.out.println("x = " + x + " y = " + y + " z = " + z);
        }
    }
```

Running `HexTest` produces the following output:

```
x = 1 y = 2147483647 z = -559035650
```

exam Watch

Don't be misled by changes in case for a hexadecimal digit or the 'x' preceding it. 0XCAFE and 0xcafe are both legal.

All three integer literals (octal, decimal, and hexadecimal) are defined as `int` by default, but they may also be specified as `long` by placing a suffix of *L* or *l* after the number:

```
long jo = 110599L;
long  so = 0xFFFFl;  // Note the lowercase 'l'
```

Floating-Point Literals

Floating-point numbers are defined as a number, a decimal symbol, and more numbers representing the fraction.

```
double d = 11301874.9881024;
```

In the preceding example, the number 11301874.9881024 is the literal value. Floating-point literals are defined as `double` (64 bits) by default, so if you want to assign a floating-point literal to a variable of type `float` (32 bits), you *must* attach the suffix *F* or *f* to the number. If you don't, the compiler will complain about a possible loss of precision, because you're trying to fit a number into a (potentially) less precise "container." The *F* suffix gives you a way to tell the compiler, "Hey, I know what I'm doing and I'll take the risk, thank you very much."

```
float f = 23.467890;  // Compiler error, possible loss of precision
float g = 49837849.029847F; // OK; has the suffix "F"
```

You may also optionally attach a *D* or *d* to double literals, but it is not necessary because this is the default behavior. But for those who enjoy typing, knock yourself out.

```
double d = 110599.995011D; // Optional, not required
double  g = 987.897; // No 'D' suffix, but OK because the
                     // literal is a double
```

exam Watch

Look for numeric literals that include a comma, for example,

```
int x = 25,343;  // Won't compile because of the comma
```

Boolean Literals

Boolean literals are the source code representation for `boolean` values. A `boolean` value can only be defined as `true` or `false`. Although in C (and some other languages) it is common to use numbers to represent `true` or `false`, *this will not work in Java.* Again, repeat after me, "Java is not C++."

```
boolean t = true; // Legal
boolean  f = 0; // Compiler error!
```

exam Watch

Be on the lookout for questions that use numbers where booleans are required. You might see an *if* ***test that uses a number, as in the following:***

```
int x = 1;  if (x) {  } // Compiler error!
```

Character Literals

A `char` literal is represented by a single character in single quotes.

```
char a = 'a';
char b = '@';
```

You can also type in the Unicode value of the character, using the Unicode notation of prefixing the value with `\u` as follows:

```
char letterN = '\u004E'; // The letter 'N'
```

exam Watch

Remember, characters are just 16-bit unsigned integers under the hood. That means you can assign a number literal, assuming it will fit into the unsigned 16-bit range (65535 or less). For example, the following are all legal:

```
char a = 0x892; // octal literal
char b = 982; // int literal
char c = (char) 70000; // The cast is required; 70000 is out of char range
char d = (char) -98; // Ridiculous, but legal
```

And the following are *not* ***legal and produce compiler errors:***

```
char e = -29; // Possible loss of precision; needs a cast
char f = 70000 // Possible loss of precision; needs a cast
```

You can also use an escape code if you want to represent a character that can't be typed in as a literal, including the characters for linefeed, newline, horizontal tab, backspace, and double and single quotes.

```
char c = '\"'; // A double quote
char  d = '\n'; // A newline
```

Now that you're familiar with the primitive data types and their ranges, you should be able to identify the proper data type to use in a given situation. Next are some examples of real-life quantities. Try to pick the primitive type that best represents the quantity.

Literal Values for Strings

A string literal is a source code representation of a value of a String object. For example, the following is an example of two ways to represent a string literal:

```
String s = "Bill Joy";
System.out.println("Bill" + " Joy");
```

SCENARIO & SOLUTION	
Which primitive type would be best to represent the number of stars in the universe?	`long`
Which primitive type would be best to represent a single multiple choice question on a test, with only one answer allowed?	`char`
Which primitive type would be best to represent a single multiple choice question on a test, with more than one answer allowed?	`char []`
Which primitive type would be best to represent the population of the U.S. in 2003?	`int` (or `long` for the world population)
Which primitive type would be best to represent the amount of money (in dollars and cents) you plan on having at retirement?	`float` (or `double` if you are a CEO of a software company)

Although strings are not primitives, they're included in this section because they can be represented as literals—in other words, *typed directly into code*. The only other nonprimitive type that has a literal representation is an array, which we'll look at in the next section.

```
Thread t = ???  // what literal value could possibly go here?
```

CERTIFICATION OBJECTIVE

Array Declaration, Construction, and Initialization (Exam Objective 1.1)

Write code that declares, constructs, and initializes arrays of any base type using any of the permitted forms both for declaration and for initialization.

Arrays are objects in Java that store multiple variables of the same type. Arrays can hold either primitives or object references, but the array itself will always be an object on the heap, even if the array is declared to hold primitive elements. In other words, there is no such thing as a primitive array, but you *can* make an array of primitives.

For this objective, you need to know three things:

- How to make an array *reference* variable (declare)
- How to make an array *object* (construct)
- How to *populate* the array with elements (initialize)

There are several different ways to do each of those, and you need to know about all of them for the exam.

Arrays are efficient, but most of the time you'll want to use one of the Collection types from java.util (including HashMap, ArrayList, TreeSet). Collection classes offer more flexible ways to access an object (for insertion, deletion, reading, etc.) and unlike arrays, can expand or contract dynamically as you add or remove elements (they're really managed arrays, since they use arrays behind the scenes). There's a Collection type for a wide range of needs. Do you need a fast sort? A group of objects with no duplicates? A way to access a name/value pair? A linked list? Chapter 6 covers them in more detail.

Declaring an Array

Arrays are declared by stating the type of element the array will hold, which can be an object or a primitive, followed by square brackets to the left or right of the identifier.

Declaring an Array of Primitives

```
int[] key; // Square brackets before name (recommended)
int key []; // Square brackets after name (legal but less readable)
```

Declaring an Array of Object References

```
Thread[] threads; // Recommended
Thread threads []; // Legal but less readable
```

on the job

When declaring an array reference, you should always put the array brackets immediately after the declared type, rather than after the identifier (variable name). That way, anyone reading the code can easily tell that, for example, *key* ***is a reference to an*** `int` *array object****, and not an*** `int` *primitive****.***

We can also declare multidimensional arrays, which are in fact arrays of arrays. This can be done in the following manner:

```
String[][][] occupantName;
String[] ManagerName [];
```

The first example is a three-dimensional array (an array of arrays of arrays) and the second is a two-dimensional array. Notice in the second example we have one square bracket before the variable name and one after. This is perfectly legal to the compiler, proving once again that *just because it's legal doesn't mean it's right.*

exam watch

It is *never* ***legal to include the size of the array in your declaration. Yes, we know you can do that in some other languages, which is why you might see a question or two that include code similar to the following:***

```
int[5] scores;
```

The preceding code won't make it past the compiler. Remember, the JVM doesn't allocate space until you actually instantiate the array object. *That's* ***when size matters.***

Constructing an Array

Constructing an array means creating the array object on the heap—in other words, doing a new on the array type. To create an array object, Java needs to know how much space to allocate on the heap, so you must specify the size of the array at construction time. The size of the array is the number of elements the array will hold.

Constructing One-Dimensional Arrays

The most straightforward way to construct an array is to use the keyword new followed by the array type, with a bracket specifying how many elements of that type the array will hold. The following is an example of constructing an array of type int:

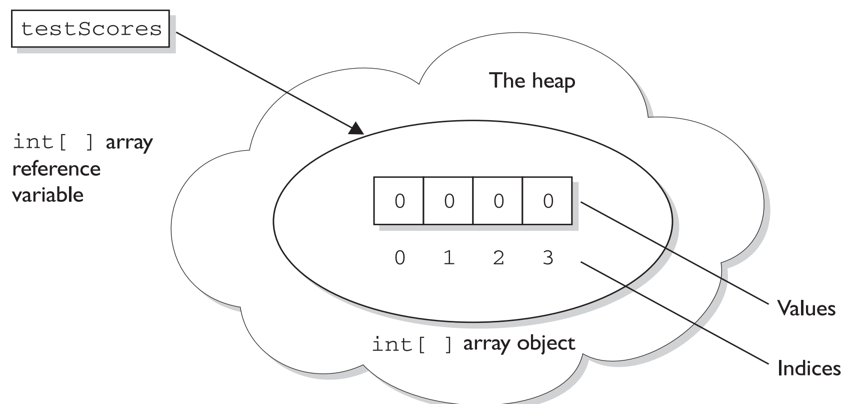
```
int[] testScores; // Declares the array of ints
testScores = new int[4]; //constructs an array and assigns it
//the testScores variable
```

The preceding code puts one new object on the heap—an array object holding four elements—with each element containing an int with a default value of 0. Think of this code as saying to the compiler, "Create an array object on the heap that will hold four primitives of type int, and assign it to the previously declared reference variable named testScores. And while you're at it, go ahead and set each int element to zero. Thanks." (The compiler appreciates good manners.) Figure 1-3 shows how the *testScores* array appears on the heap, after construction.

The next objective (4.5) covers more detail on the default values for array *elements*, but for now we're more concerned with how the *array object* itself is initialized.

FIGURE 1-3

A one-dimensional array on the heap



You can also declare and construct an array in one statement as follows:

```
int[] testScores = new int[14];
```

This single statement produces the same result as the two previous statements. Arrays of object types can be constructed in the same way:

```
Thread[] threads = new Thread[5];
```

The key point to remember here is that—despite how the code appears—*the Thread constructor is not being invoked.* We're not creating a *Thread instance,* but rather a single *Thread array* object. After the preceding statements, there are still no actual Thread objects!

exam Watch

Think carefully about how many objects are on the heap after a code statement or block executes. The exam will expect you to know, for example, that the preceding code produces just one object (the array assigned to the reference variable named** threads**). The single object referenced by** threads **holds five Thread** reference variables**, but no Thread** objects **have been created or assigned to those references.

Remember, arrays must *always* be given a size at the time they are constructed. The JVM needs the size to allocate the appropriate space on the heap for the new array object. It is never legal, for example, to do the following:

```
int[] carList = new int[]; // Will not compile; needs a size
```

So don't do it, and if you see it on the test, run screaming toward the nearest answer marked "Compilation fails."

exam Watch

You may see the words** construct**,** create**, and** instantiate **used interchangeably. They all mean, "An object is built and placed on the heap." These words also imply that the object's constructor runs, as a result of the contruct/create/instantiate code. You can say with certainty, for example, that any code that uses the keyword** `new` **will (if it runs successfully) cause the class constructor and all superclass constructors to run.

In addition to being constructed with `new`, arrays can also be created using a kind of syntax shorthand that creates the array while simultaneously initializing the array elements to values supplied in code (as opposed to default values). We'll look

at that in detail in the section on initialization. For now, understand that because of these syntax shortcuts, objects can still be created even without you ever using or seeing the keyword `new`.

Constructing Multidimensional Arrays

Multidimensional arrays, remember, are simply arrays of arrays. So a two-dimensional array of type `int` is really an object of type `int` array (`int []`), with each element in that array holding a reference to another `int` array. The second dimension holds the actual `int` primitives.

The following code declares and constructs a two-dimensional array of type `int`:

```
int[][] ratings = new int[3][];
```

Notice that only the first brackets are given a size. That's acceptable in Java, since the JVM needs to know only the size of the object assigned to the variable `ratings`.

Figure 1-4 shows how a two-dimensional `int` array works on the heap.

Initializing an Array

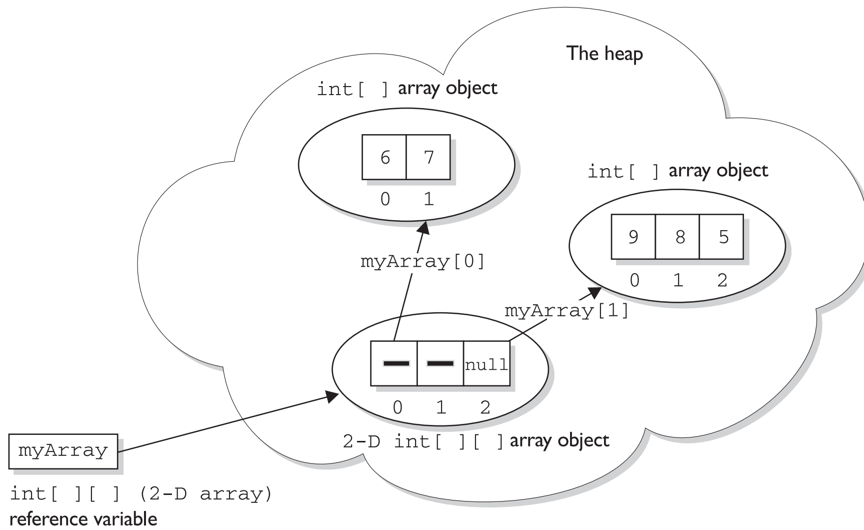
Initializing an array means putting things into it. *Things* (why, yes that *is* a technical term) in the array are the array's elements, and they're either primitive values (`2`, `'a'`, `false`, etc.), or objects referred to by the reference variables in the array. If you have an array of objects (as opposed to primitives) the array doesn't actually *hold* the objects, just as any other nonprimitive variable never actually holds the *object*, but instead holds a *reference* to the object. But we talk about arrays as, for example, "an array of five strings", even though what we really mean is, "an array of five *references* to String objects." Then the big question becomes whether or not those references are actually pointing (oops, this is Java, we mean *referring)* to real String objects, or are simply *null*. Remember, a reference that has not had an object assigned to it is a *null* reference. And if you try to actually use that *null* reference by, say, applying the dot operator to invoke a method on it, you'll get the infamous *NullPointerException*.

The individual elements in the array can be accessed with an index number. The index number always begins with zero, so for an array of ten objects the index numbers will run from 0 through 9. Suppose we create an array of three Animals as follows:

```
Animal [] pets = new Animal[3];
```

FIGURE 1-3

A two-dimensional array on the heap



Picture demonstrates the result of the following code:

```
int[ ][ ] myArray = new int[3][ ];
myArray[0] = new int[2];
myArray[0][0] = 6;
myArray[0][1] = 7;
myArray[1] = new int[3];
myArray[1][0] = 9;
myArray[1][1] = 8;
myArray[1][2] = 5;
```

We have one array object on the heap, with three *null* references of type Animal, but we still do not have any Animal objects. The next step is to create some Animal objects and assign them to index positions in the array referenced by *pets*:

```
pets[0] = new Animal();
pets[1] = new Animal();
pets[2] = new Animal();
```

This code puts three new Animal objects on the heap and assigns them to the three index positions (elements) in the *pets* array.

Look for code that tries to access an out of range array index. For example, if an array has three elements, trying to access the [3] element will raise an `ArrayIndexOutOfBoundsException`***, because in an array of three elements, the legal index values are 0, 1, and 2. You also might see an attempt to use a negative number as an array index. The following are examples of legal and illegal array access attempts. Be sure to recognize that these cause runtime exceptions and not compiler errors! Nearly all of the exam questions list both runtime exception and compiler error as possible answers.***

```
int[] x = new int[5];
x[4] = 2; // OK, the last element is at index 4
x[5] = 3; // Runtime  exception. There is no element at index 5!

int [] z = new int[2];
int y = -3;
z[y] = 4; // Runtime exception.; y is a negative number
```

These can be hard to spot in a complex loop, but that's where you're most likely to see array index problems in exam questions.

A two-dimensional array (an array of arrays) can be initialized as follows:

```
int[][] scores = new int[3][];
// Declare and create an array holding three references to int arrays

scores[0] = new int[4];
// the first element in the scores array is an int array of four int element

scores[1] = new int[6];
// the second element in the scores array is an int array of six int elements

scores[2] = new int[1];
// the third element in the scores array is an int array of one int element
```

Initializing Elements in a Loop

Array objects have a single public variable *length* that gives you the number of elements in the array. The last index value, then, is always one less than the length. For example, if the length of an array is 4, the index values are from 0 through 3. Often, you'll see array elements initialized in a loop as follows:

```
Dog[] myDogs = new Dog[6]; // creates an array of 6 Dog references
for (int x = 0; x < myDogs.length; x++) {
```

```
    myDogs[x] = new Dog(); // assign a new Dog to the index position x
}
```

The *length* variable tells us how many elements the array holds, but it does *not* tell us whether those elements have been initialized.

Declaring, Constructing, and Initializing on One Line

You can use two different array-specific syntax shortcuts to both initialize (put explicit values into an array's elements) and construct (instantiate the array object itself) in a single statement. The first is used to declare, create, and initialize in one statement as follows:

```
1.  int x = 9;
2.  int[] dots = {3,6,x,8};
```

Line 2 in the preceding code does four things:

- Declares an `int` array reference variable named `dots`.
- Creates an `int` array with a length of four (four elements).
- Populates the elements with the values 3, 6, 9, and 8.
- Assigns the new array object to the reference variable `dots`.

The size (length of the array) is determined by the number of items between the comma-separated curly braces. The code is functionally equivalent to the following longer code:

```
int[] dots;
dots = new int[4];
int x = 9;
dots[0] = 3;
dots[1] = 6;
dots[2] = x;
dots[3] = 8;
```

This begs the question, "Why would anyone use the longer way?" Two reasons come to mind. First, you might not know—at the time you create the array—the values that will be assigned to the array's elements. Second, you might just *prefer* doing it the long, slower-to-type way. Or third (OK, that's *three* reasons), maybe you just didn't know it was possible. This array shortcut alone is worth the price of this book (well, that combined with the delightful prose).

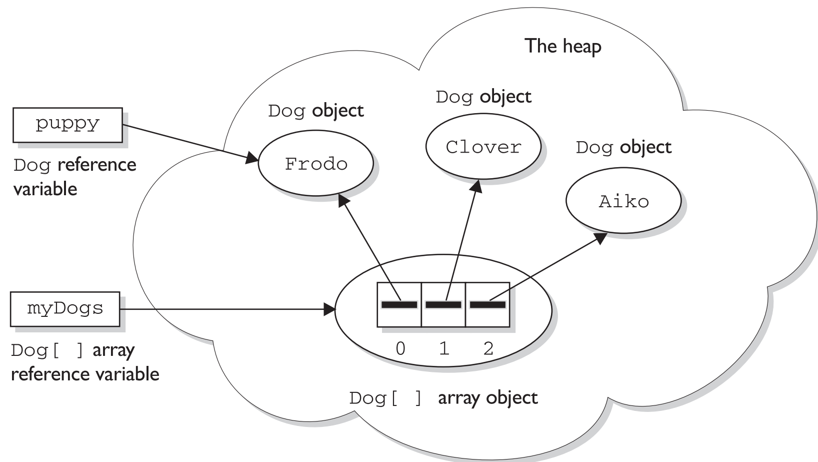
With object references rather than primitives, it works exactly the same way:

```
Dog puppy = new Dog("Frodo");
Dog[] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```

The preceding code creates one Dog array, referenced by the variable *myDogs*, with a length of three elements. It assigns a previously created Dog object (assigned to the reference variable puppy) to the first element in the array, and also creates two new Dog objects ("Clover" and "Aiko"), and assigns the two newly created instances to the last two Dog reference variable elements in the *myDogs* array. Figure 1-5 shows the result of the preceding code.

FIGURE 1-4

Declaring, constructing, and initializing an array of objects



Picture demonstrates the result of the following code:

```
Dog puppy = new Dog("Frodo");
Dog[ ] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```

Four objects are created:
1 Dog object referenced by puppy
1 Dog[ ] array referenced by myDogs
2 Dog objects referenced by myDogs[0] and myDogs[1]

You can also use the shortcut syntax with multidimensional arrays, as follows:

```
int[][] scores = {{5,2,4,7}, {9,2}, {3,4}};
```

The preceding code creates a total of four objects on the heap. First, an array of `int` arrays is constructed (the object that will be assigned to the `scores` reference variable). The *scores* array has a length of three, derived from the number of items (comma-separated) between the outer curly braces. Each of the three elements in the `scores` array is a reference variable to an `int` array, so the three `int` arrays are constructed and assigned to the three elements in the *scores* array.

The size of each of the three `int` arrays is derived from the number of items within the corresponding inner curly braces. For example, the first array has a length of four, the second array has a length of two, and the third array has a length of two. So far we have four objects: one array of `int` arrays (each element is a reference to an `int` array), and three `int` arrays (each element in the three `int` arrays is an `int` value). Finally, the three `int` arrays are initialized with the actual `int` values within the inner curly braces. Thus, the first `int` array contains the values 5, 2, 4, and 7. The following code shows the values of some of the elements in this two-dimensional array:

```
scores[0] // an array of four ints
scores[1] // an array of 2 ints
scores[2] // an array of 2 ints
scores[0][1] // the int value 5
scores[2][1] // the int value 4
```

Figure 1-6 shows the result of declaring, constructing, and initializing a two-dimensional array in one statement.

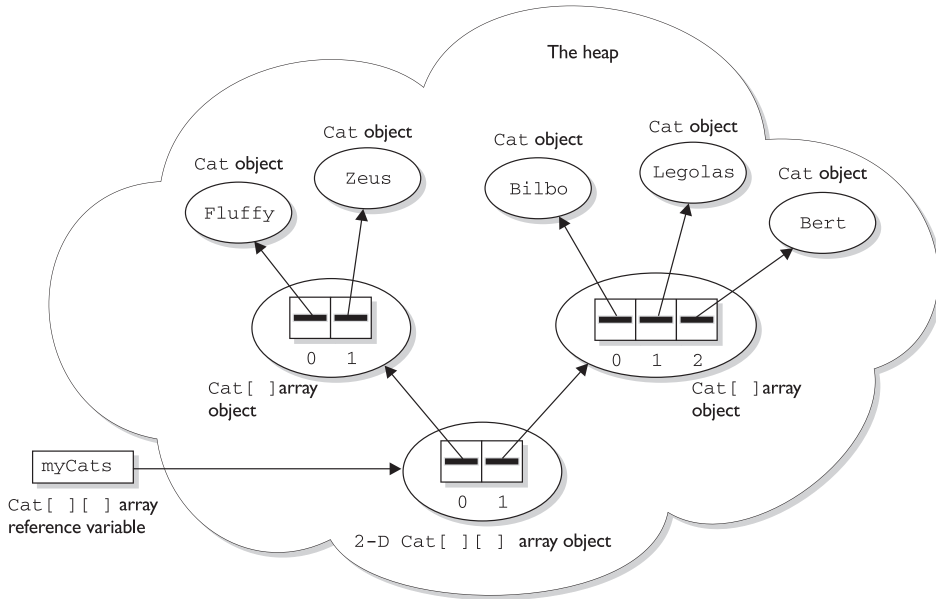
Constructing and Initializing an Anonymous Array

The second shortcut is called *anonymous array creation* and can be used to construct and initialize an array, and then assign the array to a previously declared array reference variable:

```
int[] testScores;
testScores = new int[] {4,7,2};
```

The preceding code creates a new `int` array with three elements, initializes the three elements with the values 4, 7, and 2, and then assigns the new array to the previously declared `int` array reference variable *testScores.* We call this anonymous array creation because with this syntax you don't even need to assign the new array to anything.

FIGURE 1-5 Declaring, constructing, and initializing a two-dimensional array



Picture demonstrates the result of the following code:

```
Cat[ ][ ] myCats = {{new Cat("Fluffy"), new Cat("Zeus")},
{new Cat("Bilbo"), new Cat("Legolas"), new Cat("Bert")}}
```

Eight objects are created:
1 `2-D Cat[ ][ ]` array object
2 `Cat[ ]` array objects
5 `Cat` objects

Maybe you're wondering, "What good is an array if you don't assign it to a reference variable?" You can use it to create a *just-in-time* array to use, for example, as an argument to a method that takes an array parameter. The following code demonstrates a just-in-time array argument:

```
public class Foof {
     void takesAnArray(int []  someArray) {
     // use the array parameter
     ...
      }
```

```
      public static void main (String [] args) {
         Foof f = new Foof();
         f.takesAnArray(new int[] {7,7,8,2,5}); //we need an array argument
      }
   }
```

exam Watch

Remember that you** do not specify a size **when using anonymous array creation syntax. The size is derived from the number of items (comma-separated) between the curly braces. Pay very close attention to the array syntax used in exam questions (and there will be a lot of them). You might see syntax such as

```
new Object[3] {null, new Object(), new Object()};
  // not legal;size must not be specified
```

Legal Array Element Assignments

What can you put in a particular array? For the exam, you need to know that arrays can have only one declared type (`int [ ]`, `Dog[ ]`, `String [ ]`, and so on) but that doesn't necessarily mean that only objects or primitives of the declared type can be assigned to the array elements. And what about the array reference itself? What kind of array object can be assigned to a particular array reference? For the exam, you'll need to know the answer to all of these questions. And, as if by magic, we're actually covering those very same topics in the following sections. Pay attention.

Arrays of Primitives

Primitive arrays can accept any value that can be promoted implicitly to the declared type of the array. Chapter 3 covers the rules for promotion in more detail, but for an example, an `int` array can hold any value that can fit into a 32-bit `int` variable. Thus, the following code is legal:

```
int[] weightList = new int[5];
byte b = 4;
char c = 'c';
short s = 7;
weightList[0] = b; // OK, byte is smaller than int
weightlist[1] = c;  // OK, char is smaller than int
weightList[2] = s;  // OK, short is smaller than int
```

Arrays of Object References

If the declared array type is a class, you can put objects of any subclass of the declared type into the array. For example, if Dog is a subclass of Animal, you can put both Dog objects and Animal objects into the array as follows:

```
class Car {}
class Subaru extends Car {}
class Honda extends Car {}
class Ferrari extends Car {}
Car [] myCars = {new Subaru(), new Honda(), new Ferrari()};
```

It helps to remember that the elements in a Car array are nothing more than Car reference variables. So anything that can be assigned to a Car reference variable can be legally assigned to a Car array element. Chapter 5 covers *polymorphic assignments* in more detail.

If the array is declared as an interface type, the array elements can refer to any instance of any class that implements the declared interface. The following code demonstrates the use of an interface as an array type:

```
interface Sporty {
     void beSporty();
}

class Ferrari extends Car implements Sporty {
     public void beSporty() {
      …
      // implement cool sporty method in a Ferrari-specific way
      }
}
class RacingFlats extends AthleticShoe implements Sporty {
     public void beSporty() {
      …
      // implement cool sporty method in a RacingShoe-specific way
      }
}
class GolfClub { }
class TestSportyThings {
     public static void main (String [] args) {
        Sporty[] sportyThings = new Sporty [3];
        sportyThings[0] = new Ferrari(); // OK, Ferrari implements Sporty
        sportyThings[1] = new RacingFlats();
        // OK, RacingFlats implements Sporty
        sportyThings[2] = new GolfClub();
```

```
            // Not OK; GolfClub does not implement Sporty
            // I don't care what anyone says
        }
    }
```

The bottom line is this: any object that passes the "IS-A" test for the declared array type can be assigned to an element of that array.

Array Reference Assignments for One-Dimensional Arrays

For the exam, you need to recognize legal and illegal assignments for array reference variables. We're not talking about references *in* the array (in other words, array *elements*), but rather references *to* the array object. For example, if you declare an `int` array, the reference variable you declared *can* be reassigned to any `int` array (of any size), but *cannot* be reassigned to anything that is *not* an `int` array, including an `int` value. Remember, all arrays are objects, so an `int` *array reference* cannot refer to an `int` *primitive*. The following code demonstrates legal and illegal assignments for primitive arrays:

```
int[] splats;
int[] dats = new int[4];
char[] letters = new char[5];
splats = dats; // OK, dats refers to an int array
splats = letters; // NOT OK, letters refers to a char array
```

It's tempting to assume that because a variable of type `byte`, `short`, or `char` can be explicitly promoted and assigned to an `int`, an array of any of those types could be assigned to an `int` array. You can't do that in Java, but it would be just like those cruel, heartless (but otherwise attractive) exam developers to put tricky array assignment questions in the exam.

Arrays that hold object references, as opposed to primitives, aren't as restrictive. Just as you can put a Honda object in a Car array (because Honda `extends` Car), you can assign an array of type Honda to a Car array reference variable as follows:

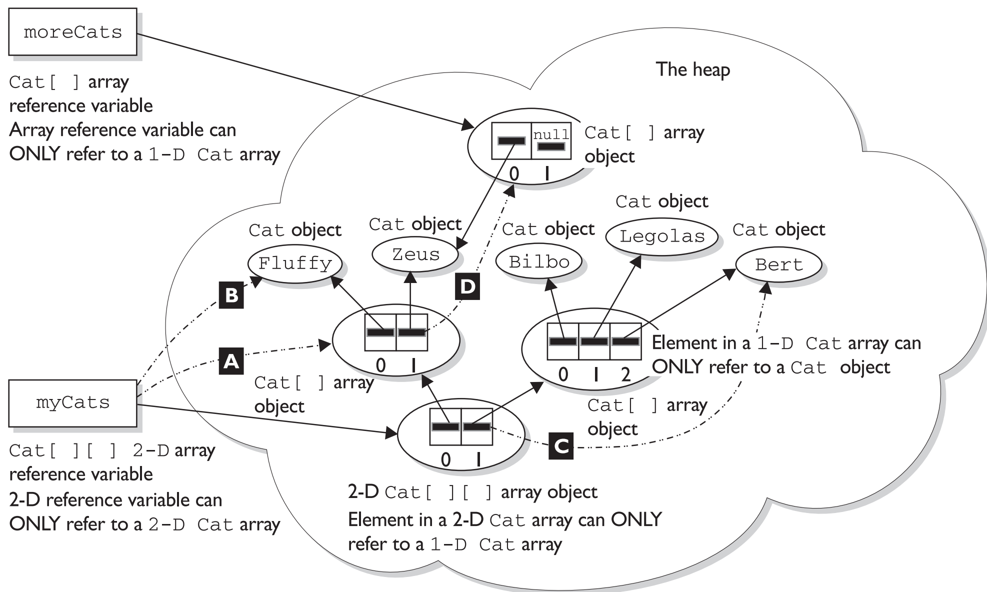
```
Car[] cars;
Honda[] cuteCars = new Honda[5];
cars = cuteCars; // OK because Honda is a type of Car
Beer[] beers = new Beer [99];
cars = beers; // NOT OK, Beer is not a type of Car
```

Apply the IS-A test to help sort the legal from the illegal. Honda IS-A Car, so a Honda array can be assigned to a Car array. Beer IS-A Car is not true; Beer does *not* extend Car (not to mention the fact that it doesn't make logical sense, unless you've already had too much of it).

exam Watch

You cannot reverse the legal assignments. ***A Car array cannot be assigned to a Honda array. A Car is not necessarily a*** `Honda`***, so if you've declared a Honda array, it might blow up if you were allowed to assign a Car array to the Honda reference variable. Think about it: a Car array could hold a reference to a Ferrari, so someone who thinks they have an array of Hondas could suddenly find themselves with a Ferrari. Remember that the IS-A test can be checked in code using the*** `instanceof` ***operator. The*** `instanceof` ***operator is covered in more detail in Chapter 3. Figure 1-7 shows an example of legal and illegal assignments for references to an array.***

FIGURE 1-6 Legal and illegal array assignments



Illegal Array Reference Assignments

A
```
myCats = myCats[0];
// Can't assign a 1-D array to a 2-D array reference
```

B
```
myCats = myCats[0][0];
// Can't assign a nonarray object to a 2-D array reference
```

C
```
myCats[1] = myCats[1][2];
// Can't assign a nonarray object to a 1-D array reference
```

D
```
myCats[0][1] = moreCats;
// Can't assign an array object to a nonarray reference
//myCats[0][1] can only refer to a Cat object
```

KEY

Legal (solid arrow)

Illegal (dashed arrow)

The rules for array assignment apply to interfaces as well as classes. An array declared as an interface type can reference an array of any type that implements the interface. Remember, any object from a class implementing a particular interface will pass the IS-A (`instanceof`) test for that interface. For example, if Box implements Foldable, the following is legal:

```
Foldable[]  foldingThings;
Box[] boxThings = new Box[3];
foldingThings = boxThings;
// OK, Box implements Foldable, so Box IS-A Foldable
```

Array Reference Assignments for Multidimensional Arrays

When you assign an array to a previously declared array reference, *the array you're assigning must be the same dimension as the reference you're assigning it to*. For example, a two-dimensional array of `int` arrays *cannot* be assigned to a regular `int` array reference, as follows:

```
int[]  blots;
int[][] squeegees = new int[3][];
blots = squeegees;  // NOT OK, squeegees is a two-d array of int arrays
int[] blocks = new int[6];
blots = blocks; // OK, blocks is an int array
```

Pay particular attention to array assignments using different dimensions. You might, for example, be asked if it's legal to assign an `int` array to the first element in an array of `int` arrays, as follows:

```
int[][]  books = new int[3][];
int[] numbers = new int[6];
int aNumber = 7;
books[0] = aNumber; //NOT OK, expecting an int array instead of an int
books[0] = numbers; //OK, numbers is an int array
```

CERTIFICATION OBJECTIVE

Using a Variable or Array Element That Is Uninitialized and Unassigned (Exam Objective 4.5)

Identify all Java programming language keywords and correctly constructed identifiers.

Java gives us the option of initializing a declared variable or leaving it uninitialized. When we attempt to *use* the uninitialized variable, we can get different behavior depending on what type of variable or array we are dealing with (primitives or objects). The behavior also depends on the level (scope) at which we are declaring our variable. An *instance variable* is declared within the class but outside any method or constructor, whereas a *local variable* is declared within a method (or in the argument list of the method).

exam Watch

Local variables are sometimes called *stack*, *temporary*, *automatic*, ***or*** *method* ***variables, but the rules for these variables are the same regardless of what you call them. Although you can leave a local variable uninitialized, the compiler complains if you try to*** *use* ***a local variable before initializing it with a value, as we shall see.***

Primitive and Object Type Instance Variables

Instance variables (also called *member variables*) are variables defined at the class level. That means the variable declaration is not made within a method, constructor, or any other initializer block. Instance variables are initialized to a default value each time a new instance is created. Table 1-3 lists the default values for primitive and object types.

Primitive Instance Variables

In the following example, the integer *year* is defined as a class member because it is within the initial curly braces of the class and not within a method's curly braces:

```
public class BirthDate {
     int year; // Instance variable
     public static void main(String [] args) {
        BirthDate bd = new BirthDate();
        bd.showYear();
     }
     public void showYear() {
        System.out.println("The year is " + year);
      }
}
```

When the program is started, it gives the variable *year* a value of zero, the default value for primitive number instance variables.

TABLE 1-3 Default Values for Primitive and Reference Types

Variable Type	Default Value
Object reference	`null` (not referencing any object)
`byte, short, int, long`	`0`
`float, double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'`

on the Job

It's a good idea to initialize all your variables, even if you're assigning them with the default value. Your code will be easier to read; programmers who have to maintain your code (after you win the lottery and move to Tahiti) will be grateful.

Object Reference Instance Variables

When compared with uninitialized primitive variables, Object references that aren't initialized are a completely different story. Let's look at the following code:

```
public class Book {
     private String title;
     public String getTitle() {
        return title;
     }
     public static void main(String [] args) {
        Book b = new Book();
        System.out.println("The title is " + b.getTitle());
     }
}
```

This code will compile fine. When we run it, the output is

```
The title is null
```

The *title* variable has not been explicitly initialized with a String assignment, so the instance variable value is *null*. Remember that *null* is not the same as an empty String (""). A *null* value means the reference variable is not referring to any object on the heap. Thus, the following modification to the Book code runs into trouble:

```
public class Book {
    private String title;
```

```
    public String getTitle() {
       return title;
    }
    public static void main(String [] args) {
       Book b = new Book();
       String s = b.getTitle(); // Compiles and runs
       String t = s.toLowerCase(); // Runtime Exception!
    }
}
```

When we try to run the Book class, the JVM will produce the following error:

```
%java Book
Exception in thread "main" java.lang.NullPointerException
     at Book.main(Book.java:12
```

We get this error because the reference variable `title` does not point (refer) to an object. We can check to see whether an object has been instantiated by using the keyword `null`, as the following revised code shows:

```
public class Book {
    private String title;
    public String getTitle() {
       return title;
    }
    public static void main(String [] args) {
       Book b = new Book();
       String s = b.getTitle(); // Compiles and runs
       if (s != null) {
          String t = s.toLowerCase();
       }
     }
}
```

The preceding code checks to make sure the object referenced by the variable `s` is not `null` before trying to use it. Watch out for scenarios on the exam where you might have to trace back through the code to find out whether an object reference will have a value of `null`. In the preceding code, for example, you look at the instance variable declaration for *title*, see that there's no explicit initialization, recognize that the *title* variable will be given the default value of `null`, and then realize that the variable *s* will also have a value of `null`. Remember, the value of *s* is a copy of the value of *title* (as returned by the `getTitle()` method), so if *title* is a *null* reference, *s* will be too.

Array Instance Variables

An array is an object; thus, an array instance variable that's declared but not explicitly initialized will have a value of `null`, just as any other object reference instance variable. But…if the array *is* initialized, what happens to the elements contained in the array? All array elements are given their default values—the same default values that elements of that type get when they're instance variables. The bottom line: *Array elements are always always always given default values, regardless of where the array itself is declared or instantiated.* By the way, if you see the word *always* three times in a row, reread the sentence three times. Now, once more, with feeling!

If we initialize an array, object reference elements will equal `null` if they are not initialized individually with values. If primitives are contained in an array, they will be given their respective default values. For example, in the following code, the array *year* will contain 100 integers that all equal zero by default:

```
public class BirthDays {
     static int [] year = new int[100];
     public static void main(String [] args) {
          for(int i=0;i<100;i++)
               System.out.println("year[" + i + "] = " + year[i]);
     }
}
```

When the preceding code runs, the output indicates that all 100 integers in the array equal zero.

Local (Stack, Automatic) Primitives and Objects

Local variables are defined within a method, including method parameters.

exam Watch

"Automatic" is just another term for "local variable." It does not mean the automatic variable is automatically assigned a value! The opposite is true; an automatic variable must be assigned a value in the code; otherwise, the compiler will complain.

Local Primitives

In the following time travel simulator, the integer *year* is defined as an automatic variable because it is within the curly braces of a method.

```
public class TimeTravel {
     public static void main(String [] args) {
          int year = 2050;
          System.out.println("The year is " + year);
     }
}
```

Okay, so we've still got work to do on the physics. *Local variables, including primitives, always always always must be initialized before you attempt to use them* (though not necessarily on the same line of code). Java does not *give* local variables a default value; *you must explicitly initialize them* with a value, as in the preceding example. If you try to use an uninitialized primitive in your code, you'll get a compiler error:

```
public class TimeTravel {
     public static void main(String [] args) {
          int year; // Local variable (declared but not initialized)
          System.out.println("The year is " + year); // Compiler error
     }
}
```

Compiling produces the following output:

```
%javac TimeTravel.java
TimeTravel.java:4: Variable year may not have been initialized.
           System.out.println("The year is " + year);
1 error
```

To correct our code, we must give the integer `year` a value. In this updated example, we declare it on a separate line, which is perfectly valid:

```
public class TimeTravel {
     public static void main(String [] args) {
          int year; // Declared but not initialized
          int day; // Declared but not initialized
          System.out.println("You step into the portal.");
          year = 2050; // Initialize (assign an explicit value)
          System.out.println("Welcome to the year " + year);
     }
}
```

Notice in the preceding example we declared an integer called *day* that never gets initialized, yet the code compiles and runs fine. Legally, you can declare a local

variable without initializing it as long as you don't *use* the variable, but let's face it, if you declared it, you probably had a reason. (Although we have heard of programmers declaring random local variables just for sport, to see if they can figure out how and why they're being used.)

The compiler can't always tell whether a local variable has been initialized before use. For example, if you initialize within a logically conditional block (in other words, a code block that may not run, such as an if block or *for* ***loop without a literal value of*** `true` ***or*** `false` ***in the test), the compiler knows that the initialization might not happen, and can produce an error. The following code upsets the compiler:***

```
public class TestLocal {
    public static void main(String [] args) {
       int x;
       if (args[0] != null) { //assume you know this will always be true
          x = 7;  // compiler can't tell that this statement will run
       }
       int y = x;
     }
}
```

The preceding code produces the following error when you attempt to compile it:

```
TestLocal.java:8: variable x might not have been initialized
  int y = x;
1 error
```

Because of the compiler-can't-tell-for-certain problem, you will sometimes need to initialize your variable outside the conditional block, just to make the compiler happy. You know why that's important if you've seen the bumper sticker: "When the compiler's not happy, ain't *nobody* happy."

Local Objects

Objects, too, behave differently when declared within a method rather than as instance variables. With instance variable object references, you can get away with leaving an object reference uninitialized, as long as the code checks to make sure the reference isn't *null* before using it. Remember, to the compiler, `null` *is* a value. You can't use the dot operator on a `null` reference, because there *is* no object at the other

end of it, but *a null reference is not the same as an uninitialized reference.* Locally declared references can't get away with checking for *null* before use, unless you explicitly initialize the local variable to `null`. The compiler will complain about the following code:

```
import java.util.Date;
public class TimeTravel {
     public static void main(String [] args) {
          Date date;
          if (date == null)
               System.out.println("date is null");
     }
}
```

Compiling the code results in the following error:

```
%javac TimeTravel.java
TimeTravel.java:5: Variable date may not have been initialized.
          If (date == null)
1 error
```

Instance variable references are always given a default value of `null`, until explicitly initialized to something else. But local references are *not* given a default value; in other words, *they aren't null.* If you don't initialize a local reference variable, then by default, its value is…well that's the whole point—it doesn't have *any* value at all! So we'll make this simple: Just set the darn thing to `null` explicitly, until you're ready to initialize it to something else. The following local variable will compile properly:

```
Date date = null; // Explicitly set the local reference variable to null
```

Local Arrays

Just like any other object reference, array references declared within a method must be assigned a value before use. That just means you must declare and construct the array. You do not, however, need to explicitly initialize the elements of an array. We've said it before, but it's important enough to repeat: *array elements are given their default values (*`0`*,* `false`*,* `null`*,* `'\u0000'`*, etc.) regardless of whether the array is declared as an instance or local variable.* The array object itself, however, will not be initialized if it's declared locally. In other words, you must explicitly initialize an array reference if it's declared and used within a method, but at the moment you construct an array object, all of its elements are assigned their default values.

CERTIFICATION OBJECTIVE

Command-Line Arguments to Main (Exam Objective 4.3)

State the correspondence between index values in the argument array passed to a main method and command line arguments.

Now that you know all about arrays, command-line arguments will be a piece of cake. Remember that the main method—the one the JVM invokes—must take a String array parameter. That String array holds the arguments you send along with the command to run your Java program, as follows:

```
class TestMain {
  public static void main (String [] args) {
     System.out.println("First arg is " + args[0]);
  }
}
```

When invoked at the command line as follows,

```
%java TestMain  Hello
```

the output is

```
First arg is Hello
```

The length of the *args* array will always be equal to the number of command-line arguments. In the following code, `args.length` is one, meaning there is one element in the array, and it is at index zero. If you try to access beyond `length-1`, you'll get an `ArrayIndexOutOfBoundsException`! This causes your entire program to explode in a spectacular JVM shutdown, so be sure the right number of arguments are being passed, perhaps with a nice user suggestion. The following code is an example of a main method expecting three arguments:

```
public static void main (String [] args) {
  if (args.length < 3) {
    System.out.println("Usage: [name] [social security #]
    //[IQ] Try again when you have a clue");
  }
}
```

exam Watch

The String array parameter does not have to be named *args* ***or*** *arg****. It can be named, for example,*** *freddie****. Also, remember that the*** *main* ***argument is just an array! There's nothing special about it, other than how it gets passed into*** *main* ***(from the JVM).***

EXERCISE 1-1

Creating a Program That Outputs Command-Line Arguments

In the following exercise…

1. Create a program that outputs every command-line argument, then displays the number of arguments.
2. You should use the array variable *length* to retrieve the length of the array.

An example of how you might write your code is at the end of this chapter.

CERTIFICATION SUMMARY

After absorbing the material in this chapter, you should be familiar with some of the nuances of the Java language. You may also be experiencing confusion around why you ever wanted to take this exam in the first place. That's normal at this point. If you hear yourself saying, "What was I thinking?" just lie down until it passes. We would *like* to tell you that it gets easier… that this was the toughest chapter and it's all downhill from here.

Let's briefly review what you'll need to know for the exam.

There will be more than one question dealing with keywords, so be sure you can identify which are keywords and which aren't. Make sure you're familiar with the ranges of integer primitives, and the bit depth of all primitives. And, although this isn't Java language specific, you must be able to convert between octal, decimal, and hexadecimal literals. You have also learned about arrays, and how they behave when declared in a class or a method.

Be certain that you know the effects of leaving a variable uninitialized, and how the variable's scope changes the behavior. You'll also be expected to know what happens to the elements of an array when they're not explicitly initialized.

For the exam, knowing what you can't do with the Java language is just as important as knowing what you can do. Give the sample questions a try! They're very similar to the difficulty and structure of the real exam questions, and should be an eye opener for how difficult the exam can be. Don't worry if you get a lot of them wrong. If you find a topic that you are weak in, spend more time reviewing and studying. Many programmers need two or three serious passes through a chapter (or an individual objective) before they can answer the questions confidently.

TWO-MINUTE DRILL

Java Programming Language Keywords

- ❑ Keywords cannot be used as identifiers (names) for classes, methods, variables, or anything else in your code.
- ❑ All keywords start with a lowercase letter.

Literals and Ranges of All Primitive Data Types

- ❑ All six number types in Java are signed, so they can be positive or negative.
- ❑ Use the formula $-2^{(bits-1)}$ to $2^{(bits-1)}-1$ to determine the range of an integer type.
- ❑ A `char` is really a 16-bit unsigned integer.
- ❑ Literals are source code representations of primitive data types, or String.
- ❑ Integers can be represented in octal (0127), decimal (1245), and hexadecimal (0XCAFE).
- ❑ Numeric literals cannot contain a comma.
- ❑ A `char` literal can be represented as a single character in single quotes (‘A’).
- ❑ A `char` literal can also be represented as a Unicode value (‘\u0041’).
- ❑ A `char` literal can also be represented as an integer, as long as the integer is less than 65536.
- ❑ A `boolean` literal can be either `true` or `false`.
- ❑ Floating-point literals are always double by default; if you want a `float`, you must append an *F* or *f* to the literal.

Array Declaration, Construction, and Initialization

- ❑ Arrays can hold primitives or objects, but the array itself is *always* an object.
- ❑ When you declare an array, the brackets can be to the left or right of the variable name.
- ❑ It is never legal to include the size of an array in the declaration.

- You must include the size of an array when you construct it (using `new`) unless you are creating an anonymous array.
- Elements in an array of objects are not automatically created, although primitive array elements are given default values.
- You'll get a `NullPointerException` if you try to use an array element in an object array, if that element does not refer to a real object.
- Arrays are indexed beginning with zero. In an array with three elements, you can access element 0, element 1, and element 2.
- You'll get an `ArrayIndexOutOfBoundsException` if you try to access outside the range of an array.
- Arrays have a *length* variable that contains the number of elements in the array.
- The last index you can access is always one less than the length of the array.
- Multidimensional arrays are just arrays of arrays.
- The dimensions in a multidimensional array can have different lengths.
- An array of primitives can accept any value that can be promoted implicitly to the declared type of the array. For example, a `byte` variable can be placed in an `int` array.
- An array of objects can hold any object that passes the IS-A (or `instanceof`) test for the declared type of the array. For example, if Horse extends Animal, then a Horse object can go into an Animal array.
- If you assign an array to a previously declared array reference, the array you're assigning must be the same dimension as the reference you're assigning it to.
- You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a Honda array can be assigned to an array declared as type Car (assuming Honda `extends` Car).

Using a Variable or Array Element That Is Uninitialized and Unassigned

- When an array of objects is instantiated, objects within the array are not instantiated automatically, but all the references get the default value of `null`.
- When an array of primitives is instantiated, all elements get their default values.

- Just as with array elements, instance variables are always initialized with a default value.
- Local/automatic/method variables are never given a default value. If you attempt to use one before initializing it, you'll get a compiler error.

Command-Line Arguments to Main

- Command-line arguments are passed to the String array parameter in the *main* method.
- The first command-line argument is the first element in the main String array parameter.
- If no arguments are passed to *main*, the length of the *main* String array parameter will be zero.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully! These questions are *very* similar to the kinds of questions you'll see on the latest exam. Again, don't worry if you have trouble with them at first; the style of the exam questions can take some getting used to. For example, you might find yourself looking at the answers and wanting to kick yourself for missing little things that you actually knew, but just didn't see in the question. The best advice we have for both the practice questions and the real exam is to *always look again*. As soon as you get an idea in your head about the answer to a question, imagine someone standing next to you and whispering in your ear, "Are you sure? Look again." Much of the time, you'll look again and say, "I'm sure," especially since your first reaction is often the best one to go with. But you'll be surprised by how often that second look brings up something new.

Java Programming Language Keywords (Objective 4.4)

1. Given the following,

```
1.  public class Test {
2.     public static void main(String [] args) {
3.        signed int x = 10;
4.           for (int y=0; y<5; y++, x--)
5.              System.out.print(" " + x);
6.     }
7.  }
```

what is the result? (Choose one.)

A. `10 9 8 7 6`

B. `9 8 7 6 5`

C. Compilation fails

D. An exception is thrown at runtime

2. Which is a reserved word in the Java programming language? (Choose one.)

A. `method`

B. `native`

C. `subclasses`

D. `reference`

E. `array`

3. Which one of these lists contains only Java programming language keywords? (Choose one.)

   A. `class`, `if`, `void`, `long`, `Int`, `continue`

   B. `goto`, `instanceof`, `native`, `finally`, `default`, `throws`

   C. `try`, `virtual`, `throw`, `final`, `volatile`, `transient`

   D. `strictfp`, `constant`, `super`, `implements`, `do`

   E. `byte`, `break`, `assert`, `switch`, `include`

4. Which two are keywords? (Choose two.)

   A. `interface`

   B. `unsigned`

   C. `Float`

   D. `this`

   E. `string`

Literals and Ranges of All Primitive Data Types (Objective 4.6)

5. Which three are valid declarations of a `char`? (Choose three.)

   A. `char c1 = 064770;`

   B. `char c2 = 'face';`

   C. `char c3 = 0xbeef;`

   D. `char c4 = \u0022;`

   E. `char c5 = '\iface';`

   F. `char c6 = '\uface';`

6. Which two are valid declarations of a String? (Choose two.)

   A. `String s1 = null;`

   B. `String s2 = 'null';`

   C. `String s3 = (String) 'abc';`

   D. `String s4 = (String) '\ufeed';`

   E. `String s5 = "strings rule";`

7. Which one is a valid declaration of a `boolean`? (Choose one.)

A. `boolean b1 = 0;`

B. `boolean b2 = 'false';`

C. `boolean b3 = false;`

D. `boolean b4 = Boolean.false();`

E. `boolean b5 = no;`

8. What is the numerical range of a `char`? (Choose one.)

A. –128 to 127

B. –(2 ^ 15) to (2 ^ 15) - 1

C. 0 to 32767

D. Platform dependent

E. 0 to 65535

9. Which three are valid declarations of a `float`? (Choose three.)

A. `float f1 = -343;`

B. `float f2 = 3.14;`

C. `float f3 = 0x12345;`

D. `float f4 = 42e7;`

E. `float f5 = 2001.0D;`

F. `float f6 = 2.81F;`

Array Declaration, Construction, and Initialization (Objective 1.1)

10. Which three are legal array declarations? (Choose three.)

A. `int [] myScores [];`

B. `char [] myChars;`

C. `int [6] myScores;`

D. `Dog myDogs [];`

E. `Dog myDogs [7];`

11. Given the following,

```
1.  public class Test {
2.     public static void main(String [] args) {
```

```
 3.     int [] [] [] x = new int [3] [] [];
 4.     int i,j;
 5.     x[0] = new int[4][];
 6.     x[1] = new int[2][];
 7.     x[2] = new int[5][];
 8.     for (i=0; i<x.length; i++)
 9.        for (j=0; j<x[i].length; j++) {
10.           x[i][j] = new int [i + j + 1];
11.           System.out.println("size = " + x[i][j].length);
12.        }
13.     }
14.  }
```

how many lines of output will be produced? (Choose one.)

A. 7

B. 9

C. 11

D. 13

E. Compilation fails

F. An exception is thrown at runtime

12. Given the following,

```
1.  public class Test {
2.     public static void main(String [] args) {
3.        byte [][] big = new byte [7][7];
4.        byte [][] b = new byte [2][1];
5.        byte b3 = 5;
6.        byte b2 [][][][] = new byte [2][3][1][2];
7.
8.     }
9.  }
```

which of the following lines of code could be inserted at line 7, and still allow the code to compile? (Choose four that would work.)

A. `b2[0][1] = b;`

B. `b[0][0] = b3;`

C. `b2[1][1][0] = b[0][0];`

D. `b2[1][2][0] = b;`

E. `b2[0][1][0][0] = b[0][0];`

F. `b2[0][1] = big;`

13. Which two will declare an array and initialize it with five numbers? (Choose two.)

A. `Array a = new Array(5);`

B. `int [] a = {23,22,21,20,19};`

C. `int [] array;`

D. `int array [] = new int [5];`

E. `int a [] = new int(5);`

F. `int [5] array;`

14. Which will legally declare, construct, and initialize an array? (Choose one.)

A. `int [] myList = {"1", "2", "3"};`

B. `int [] myList = (5, 8, 2);`

C. `int myList [] [] = {4,9,7,0};`

D. `int myList [] = {4, 3, 7};`

E. `int [] myList = [3, 5, 6];`

F. `int myList [] = {4; 6; 5};`

Using a Variable or Array Element That Is Uninitialized and Unassigned (Objective 4.5)

15. Which four describe the correct default values for array elements of the types indicated? (Choose four.)

A. `int -> 0`

B. `String -> "null"`

C. `Dog -> null`

D. `char -> '\u0000'`

E. `float -> 0.0f`

F. `boolean -> true`

16. Given the following,

```
1.  public class TestDogs {
2.     public static void main(String [] args) {
3.        Dog [][] theDogs = new Dog[3][];
4.        System.out.println(theDogs[2][0].toString());
5.     }
6.  }
```

```
7.
8.  class Dog {}
```

what is the result? (Choose one.)

A. `null`

B. `theDogs`

C. Compilation fails

D. An exception is thrown at runtime

17. Given the following,

```
1.  public class X {
2.     public static void main(String [] args) {
3.        String names [] = new String[5];
4.           for (int x=0; x < args.length; x++)
5.              names[x] = args[x];
6.           System.out.println(names[2]);
7.     }
8.  }
```

and the command line invocation is

```
java X a b
```

what is the result? (Choose one.)

A. `names`

B. `null`

C. Compilation fails

D. An exception is thrown at runtime

Command-Line Arguments to Main (Objective 4.3)

18. Given the following,

```
1.  public class CommandArgs {
2.     public static void main(String [] args) {
3.        String s1 = args[1];
4.        String s2 = args[2];
5.        String s3 = args[3];
6.        String s4 = args[4];
7.        System.out.print(" args[2] = " + s2);
8.     }
9.  }
```

and the command-line invocation,

```
java CommandArgs 1 2 3 4
```

what is the result?

A. `args[2] = 2`

B. `args[2] = 3`

C. `args[2] = null`

D. `args[2] = 1`

E. Compilation fails

F. An exception is thrown at runtime

19. Given the following,

```
 1.  public class CommandArgsTwo {
 2.     public static void main(String [] argh) {
 3.        String [] args;
 4.        int x;
 5.        x = argh.length;
 6.        for (int y = 1; y <= x; y++) {
 7.           System.out.print(" " + argh[y]);
 8.        }
 9.     }
10.  }
```

and the command-line invocation,

```
java CommandArgsTwo 1 2 3
```

what is the result?

A. `0 1 2`

B. `1 2 3`

C. `0 0 0`

D. `null null null`

E. Compilation fails

F. An exception is thrown at runtime

20. Given the following,

```
 1.  public class CommandArgsThree {
 2.     public static void main(String [] args) {
 3.        String [][] argCopy = new String[2][2];
 4.        int x;
 5.        argCopy[0] = args;
 6.        x = argCopy[0].length;
 7.        for (int y = 0; y < x; y++) {
 8.           System.out.print(" " + argCopy[0][y]);
 9.        }
10.     }
11.  }
```

and the command-line invocation,

```
java CommandArgsThree 1 2 3
```

what is the result?

A. `0 0`

B. `1 2`

C. `0 0 0`

D. `1 2 3`

E. Compilation fails

F. An exception is thrown at runtime

SELF TEST ANSWERS

Java Programming Language Keywords (Objective 4.4)

1. ☑ **C.** The word "signed" is not a valid modifier keyword in the Java language. All number primitives in Java are signed. Always.

2. ☑ **B.** The word `native` is a valid keyword, used to modify a method declaration.
☒ **A**, **D**, and **E** are not keywords. **C** is wrong because the keyword for subclassing in Java is `extends`, not 'subclasses'.

3. ☑ **B.** All the words in answer **B** are among the 49 Java keywords.
☒ **A** is wrong because the keyword for the primitive `int` starts with a lowercase *i*. **C** is wrong because "virtual" is a keyword in C++, but not Java. **D** is wrong because "constant" is not a keyword. Constants in Java are marked `static` and `final`. **E** is wrong because "include" is a keyword in C, but not Java.

4. ☑ **A** and **D.** Both `interface` and `this` are both valid keywords.
☒ **B** is wrong because "unsigned" is a keyword in C/C++ but not in Java. **C** is wrong because "Float" is a class type. The keyword for the Java primitive is `float`. **E** is wrong because although "String" is a class type in Java, "string" is not a keyword.

Literals and Ranges of All Primitive Data Types (Objective 4.6)

5. ☑ **A, C,** and **F. A** is an octal representation of the integer value 27128, which is legal because it fits into an unsigned 16-bit integer. **C** is a hexadecimal representation of the integer value 48879, which fits into an unsigned 16-bit integer. **F** is a Unicode representation of a character.
☒ **B** is wrong because you can't put more than one character in a `char` literal. You know that **B** is a literal character because it comes between single quotes. The only other acceptable `char` literal that can go between single quotes is a Unicode value, and Unicode literals must always start with a '`\u`'. **D** is wrong because the single quotes are missing. **E** is wrong because it appears to be a Unicode representation (notice the backslash), but starts with '`\i`' rather than '`\u`'.

6. ☑ **A** and **E. A** sets the String reference to null; **E** initializes the String reference with a literal.
☒ **B** is wrong because null cannot be in single quotes. **C** is wrong because there are multiple characters between the single quotes ('abc'). **D** is wrong because you can't cast a `char` (primitive) to a String (object).

7. ☑ **C.** A boolean can only be assigned the literal `true` or `false`.
☒ **A, B, D,** and **E** are all invalid assignments for a boolean.

8. ☑ **E.** A `char` is really a 16-bit integer behind the scenes, so it supports 2^{16} (from 0 to 65535) values.

9. ☑ **A, C,** and **F. A** and **C** are integer literals (32 bits), and integers can be legally assigned to floats (also 32 bits). **F** is correct because *F* is appended to the literal, declaring it as a `float` rather than a `double` (the default for floating point literals).
☒ **B**, **D**, and **E** are all doubles.

Array Declaration, Construction, and Initialization (Objective 1.1)

10. ☑ **A, B,** and **D.** With an array declaration, you can place the brackets to the right or left of the identifier. **A** looks strange, but it's perfectly legal to split the brackets in a multidimensional array, and place them on both sides of the identifier. Although coding this way would only annoy your fellow programmers, for the exam, you need to know it's legal.
☒ **C** and **E** are wrong because you can't declare an array with a size. The size is only needed when the array is actually instantiated (and the JVM needs to know how much space to allocate for the array, based on the type of array and the size).

11. ☑ **C.** The loops use the array sizes (length).
If you think this question is unfairly complicated, get used to it. Question 11 is a good example of the kinds of questions you'll see on the exam. You should approach complex loop questions by using a pencil and paper and stepping through the loop (or loops, in this case), keeping track of the variable values at each iteration. Tedious, we know, but you can expect a lot of questions like this on the exam. Take your time and recheck your work.

12. ☑ **A, B, E,** and **F.** This question covers the issue of, "What can I assign to an array reference variable?" The key is to get the dimensions right. For example, if an array is declared as a two-dimensional array, you can't assign a one-dimensional array to a one-dimensional array reference.
☒ **C** is wrong because it tries to assign a primitive byte where a byte *array* (one dimension) is expected. **D** is wrong because it tries to assign a two-dimensional array where a one-dimensional array is expected.

13. ☑ **B** and **D.** Both are legal ways to declare and initialize an array with five elements.
☒ **A** is wrong because it shows an example of instantiating a *class* named Array, passing the integer value 5 to the object's constructor. If you don't see the brackets, you can be

certain there is no actual array object! In other words, an Array object (instance of class Array) is not the same as an array object. **C** is wrong because it shows a legal array declaration, but with no initialization. **E** is wrong (and will not compile) because the initialization uses parens () rather than brackets. **F** is wrong (and will not compile) because it declares an array with a size. Arrays must never be given a size when declared.

14. ☑ **D.** The only legal array declaration and assignment statement is **D**.
☒ **A** is wrong because it initializes an `int` array with String literals. **B** and **E** are wrong because they use something other than curly braces for the initialization. **C** is wrong because it provides initial values for only one dimension, although the declared array is a two-dimensional array. **F** is wrong because it uses semicolons where it should use commas, to separate the items in the initialization.

Using a Variable or Array Element That Is Uninitialized and Unassigned (Objective 4.5)

15. ☑ A, C, D, and E.
☒ **B** is wrong because the default value for a String (and any other object reference) is `null`, with no quotes. **F** is wrong because the default value for `boolean` elements is `false`.

16. ☑ **D.** The second dimension of the array referenced by `theDogs` has not been initialized. Attempting to access an uninitialized object element (line 4) raises a `NullPointerException`.

17. ☑ **B.** The *names* array is initialized with five *null* elements. Then elements 0 and 1 are assigned the String values "a" and "b" respectively (the command-line arguments passed to main). Elements 2, 3, and 4 remain unassigned, so they have a value of `null`.

Command-line Arguments to Main (Objective 4.3)

18. ☑ **F.** An exception is thrown because at line 6, the array index (the fifth element) is out of bounds. The exception thrown is the cleverly named `ArrayIndexOutOfBoundsException`.

19. ☑ **F.** An exception is thrown because at some point in line 7, the value of x will be equal to y, resulting in an attempt to access an index out of bounds for the array. Remember that you can access only as far as *length*-1, so loop logical tests should use `x<someArray.length` as opposed to `x <= someArray.length`.

20. ☑ **D.** In line 5, the reference variable `argCopy[0]`, which was referring to an array with two elements, is reassigned to an array (args) with three elements.

EXERCISE ANSWERS

Exercise 1.1: Command-Line Arguments to Main

Your completed code should look something like the following:

```
public class MainTest {
     public static void main (String [] args) {
          for (int i = 0;i < args.length;i++) {
               System.out.println(args[i]);
          }
          System.out.println("Total words:  " + args.length);
     }
}
```