3

Assignments

CERTIFICATION OBJECTIVES

- Use Class Members
- Develop Wrapper Code & Autoboxing Code
- Determine the Effects of Passing Variables into Methods
- Recognize when Objects Become Eligible for Garbage Collection

✓ Two-Minute Drill

Q&A Self Test

Stack and Heap—Quick Review

For most people, understanding the basics of the stack and the heap makes it far easier to understand topics like argument passing, polymorphism, threads, exceptions, and garbage collection. In this section, we'll stick to an overview, but we'll expand these topics several more times throughout the book.

For the most part, the various pieces (methods, variables, and objects) of Java programs live in one of two places in memory: the stack or the heap. For now, we're going to worry about only three types of things: instance variables, local variables, and objects:

- Instance variables and objects live on the heap.
- Local variables live on the stack.

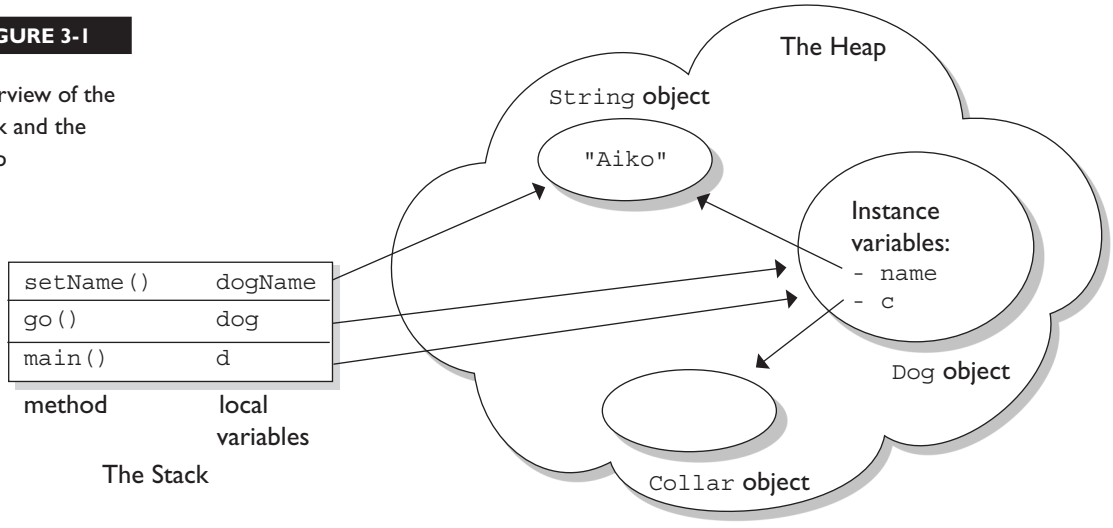
Let's take a look at a Java program, and how its various pieces are created and map into the stack and the heap:

```
 1. class Collar { }
 2.
 3. class Dog {
 4.   Collar c;         // instance variable
 5.   String name;      // instance variable
 6.
 7.   public static void main(String [] args) {
 8.
 9.     Dog d;                          // local variable: d
10.     d = new Dog();
11.     d.go(d);
12.   }
13.   void go(Dog dog) {                // local variable: dog
14.     c = new Collar();
15.     dog.setName("Fido");
16.   }
17.   void setName(String dogName) {    // local var: dogName
18.     name = dogName;
19.     // do more stuff
20.   }
21. }
```

Figure 3-1 shows the state of the stack and the heap once the program reaches line 19. Following are some key points:

FIGURE 3-1

Overview of the Stack and the Heap



- Line 7—`main()` is placed on the stack.
- Line 9—reference variable `d` is created on the stack, but there's no `Dog` object yet.
- Line 10—a new `Dog` object is created and is assigned to the `d` reference variable.
- Line 11—a copy of the reference variable `d` is passed to the `go()` method.
- Line 13—the `go()` method is placed on the stack, with the `dog` parameter as a local variable.
- Line 14—a new `Collar` object is created on the heap, and assigned to Dog's instance variable.
- Line 17—`setName()` is added to the stack, with the `dogName` parameter as its local variable.
- Line 18—the `name` instance variable now also refers to the String object.
- Notice that two *different* local variables refer to the same `Dog` object.
- Notice that one local variable and one instance variable both refer to the same String `Aiko`.
- After Line 19 completes, `setName()` completes and is removed from the stack. At this point the local variable `dogName` disappears too, although the String object it referred to is still on the heap.

CERTIFICATION OBJECTIVE

Literals, Assignments, and Variables (Exam Objectives 1.3 and 7.6)

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.

7.6 Write code that correctly applies the appropriate operators including assignment operators (limited to: =, +=, -=)…

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.

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 six = 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 allowed only 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
```

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

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.

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

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!
```

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'
```

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;         // hexadecimal 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 single quotes.

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

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");
```

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 later in the chapter.

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

Assignment Operators

Assigning a value to a variable seems straightforward enough; you simply assign the stuff on the right side of the = to the variable on the left. Well, sure, but don't expect to be tested on something like this:

```
x = 6;
```

No, you won't be tested on the no-brainer (technical term) assignments. You will, however, be tested on the trickier assignments involving complex

expressions and casting. We'll look at both primitive and reference variable assignments. But before we begin, let's back up and peek inside a variable. What is a variable? How are the variable and its value related?

Variables are just bit holders, with a designated type. You can have an `int` holder, a `double` holder, a Button holder, and even a String[] holder. Within that holder is a bunch of bits representing the value. For primitives, the bits represent a numeric value (although we don't know what that bit pattern looks like for `boolean`, luckily, we don't care). A `byte` with a value of 6, for example, means that the bit pattern in the variable (the `byte` holder) is 00000110, representing the 8 bits.

So the value of a primitive variable is clear, but what's inside an object holder? If you say,

```
Button b = new Button();
```

what's inside the Button holder `b`? Is it the Button object? No! A variable referring to an object is just that—a *reference* variable. A reference variable bit holder contains bits representing a *way to get to the object*. We don't know what the format is. The way in which object references are stored is virtual-machine specific (it's a pointer to something, we just don't know what that something really is). All we can say for sure is that the variable's value is *not* the object, but rather a value representing a specific object on the heap. Or `null`. If the reference variable has not been assigned a value, or has been explicitly assigned a value of `null`, the variable holds bits representing—you guessed it—`null`. You can read

```
Button b = null;
```

as "The Button variable b is not referring to any object."

So now that we know a variable is just a little box o' bits, we can get on with the work of changing those bits. We'll look first at assigning values to primitives, and finish with assignments to reference variables.

Primitive Assignments

The equal (=) sign is used for assigning a value to a variable, and it's cleverly named the assignment operator. There are actually 12 assignment operators, but only the five most commonly used are on the exam, and they are covered in Chapter 4.

You can assign a primitive variable using a literal or the result of an expression.

Take a look at the following:

```
int x = 7;      // literal assignment
int y = x + 2; // assignment with an expression
               // (including a literal)
int z = x * y; // assignment with an expression
```

The most important point to remember is that a literal integer (such as 7) is always implicitly an `int`. Thinking back to Chapter 1, you'll recall that an `int` is a 32-bit value. No big deal if you're assigning a value to an `int` or a `long` variable, but what if you're assigning to a `byte` variable? After all, a `byte`-sized holder can't hold as many bits as an `int`-sized holder. Here's where it gets weird. The following is legal,

```
byte b = 27;
```

but only because the compiler automatically narrows the literal value to a `byte`. In other words, the compiler puts in the *cast*. The preceding code is identical to the following:

```
byte b = (byte) 27; // Explicitly cast the int literal to a byte
```

It looks as though the compiler gives you a break, and lets you take a shortcut with assignments to integer variables smaller than an `int`. (Everything we're saying about `byte` applies equally to `char` and `short`, both of which are smaller than an `int`.) We're not actually at the weird part yet, by the way.

We know that a literal integer is always an `int`, but more importantly, the result of an expression involving anything `int`-sized or smaller is always an `int`. In other words, add two `bytes` together and you'll get an `int`—even if those two bytes are tiny. Multiply an `int` and a `short` and you'll get an `int`. Divide a `short` by a `byte` and you'll get…an `int`. OK, now we're at the weird part. Check this out:

```
byte b = 3;      // No problem, 3 fits in a byte
byte c = 8;      // No problem, 8 fits in a byte
byte d = b + c; // Should be no problem, sum of the two bytes
                // fits in a byte
```

The last line won't compile! You'll get an error something like this:

```
TestBytes.java:5: possible loss of precision
found   : int
required: byte
    byte c = a + b;
             ^
```

We tried to assign the sum of two bytes to a `byte` variable, the result of which (11) was definitely small enough to fit into a `byte`, but the compiler didn't care. It knew the rule about `int`-or-smaller expressions always resulting in an `int`. It would have compiled if we'd done the *explicit* cast:

```
byte c = (byte) (a + b);
```

Primitive Casting

Casting lets you convert primitive values from one type to another. We mentioned primitive casting in the previous section, but now we're going to take a deeper look. (Object casting was covered in Chapter 2.)

Casts can be implicit or explicit. An implicit cast means you don't have to write code for the cast; the conversion happens automatically. Typically, an implicit cast happens when you're doing a widening conversion. In other words, putting a smaller thing (say, a `byte`) into a bigger container (like an `int`). Remember those "`possible loss of precision`" compiler errors we saw in the assignments section? Those happened when we tried to put a larger thing (say, a `long`) into a smaller container (like a `short`). The large-value-into-small-container conversion is referred to as *narrowing* and requires an explicit cast, where you tell the compiler that you're aware of the danger and accept full responsibility. First we'll look at an implicit cast:

```
int a = 100;
long b = a; // Implicit cast, an int value always fits in a long
```

An explicit casts looks like this:

```
float a = 100.001f;
int b = (int)a; // Explicit cast, the float could lose info
```

Integer values may be assigned to a `double` variable without explicit casting, because any integer value can fit in a 64-bit `double`. The following line demonstrates this:

```
double d = 100L; // Implicit cast
```

In the preceding statement, a `double` is initialized with a `long` value (as denoted by the `L` after the numeric value). No cast is needed in this case because a `double` can hold every piece of information that a `long` can store. If, however, we want to assign a `double` value to an integer type, we're attempting a narrowing conversion and the compiler knows it:

```
class Casting {
  public static void main(String [] args) {
    int x = 3957.229; // illegal
  }
}
```

If we try to compile the preceding code, we get an error something like:

```
%javac Casting.java
Casting.java:3: Incompatible type for declaration. Explicit cast
needed to convert double to int.
      int x = 3957.229; // illegal
1 error
```

In the preceding code, a floating-point value is being assigned to an integer variable. Because an integer is not capable of storing decimal places, an error occurs. To make this work, we'll cast the floating-point number into an `int`:

```
class Casting {
  public static void main(String [] args) {
    int x = (int)3957.229; // legal cast
    System.out.println("int x = " + x);
  }
}
```

When you cast a floating-point number to an integer type, the value loses all the digits after the decimal. The preceding code will produce the following output:

```
int x = 3957
```

We can also cast a larger number type, such as a `long`, into a smaller number type, such as a `byte`. Look at the following:

```
class Casting {
  public static void main(String [] args) {
    long l = 56L;
    byte b = (byte)l;
    System.out.println("The byte is " + b);
  }
}
```

The preceding code will compile and run fine. But what happens if the `long` value is larger than 127 (the largest number a `byte` can store)? Let's modify the code:

```
class Casting {
  public static void main(String [] args) {
    long l = 130L;
    byte b = (byte)l;
    System.out.println("The byte is " + b);
  }
}
```

The code compiles fine, and when we run it we get the following:

```
%java Casting
The byte is -126
```

You don't get a runtime error, even when the value being narrowed is too large for the type. The bits to the left of the lower 8 just…go away. If the leftmost bit (the sign bit) in the `byte` (or any integer primitive) now happens to be a 1, the primitive will have a negative value.

EXERCISE 3-1

Casting Primitives

Create a `float` number type of any value, and assign it to a `short` using casting.

1. Declare a `float` variable: `float f = 234.56F;`
2. Assign the `float` to a `short`: `short s = (short)f;`

Assigning Floating-Point Numbers Floating-point numbers have slightly different assignment behavior than integer types. First, you must know that every floating-point literal is implicitly a `double` (64 bits), not a `float`. So the literal `32.3`, for example, is considered a `double`. If you try to assign a `double` to a `float`, the compiler knows you don't have enough room in a 32-bit `float` container to hold the precision of a 64-bit `double`, and it lets you know. The following code looks good, but won't compile:

```
float f = 32.3;
```

You can see that `32.3` should fit just fine into a float-sized variable, but the compiler won't allow it. In order to assign a floating-point literal to a `float` variable, you must either cast the value or append an `f` to the end of the literal. The following assignments will compile:

```
float f = (float) 32.3;
float g = 32.3f;
float h = 32.3F;
```

Assigning a Literal That Is Too Large for the Variable We'll also get a compiler error if we try to assign a literal value that the compiler knows is too big to fit into the variable.

```
byte a = 128; // byte can only hold up to 127
```

The preceding code gives us an error something like

```
TestBytes.java:5: possible loss of precision
found   : int
required: byte
byte a = 128;
```

We can fix it with a cast:

```
byte a = (byte) 128;
```

But then what's the result? When you narrow a primitive, Java simply truncates the higher-order bits that won't fit. In other words, it loses all the bits to the left of the bits you're narrowing to.

Let's take a look at what happens in the preceding code. There, 128 is the bit pattern 10000000. It takes a full 8 bits to represent 128. But because the literal 128 is an int, we actually get 32 bits, with the 128 living in the right-most (lower-order) 8 bits. So a literal 128 is actually

00000000000000000000000010000000

Take our word for it; there are 32 bits there.

To narrow the 32 bits representing 128, Java simply lops off the leftmost (higher-order) 24 bits. We're left with just the 10000000. But remember that a byte is signed, with the leftmost bit representing the sign (and not part of the value of the variable). So we end up with a negative number (the 1 that used to represent 128 now represents the negative sign bit). Remember, to find out the value of a negative number using two's complement notation, you flip all of the bits and then add 1. Flipping the 8 bits gives us 01111111, and adding 1 to that gives us 10000000, or back to 128! And when we apply the sign bit, we end up with -128.

You must use an explicit cast to assign 128 to a byte, and the assignment leaves you with the value -128. A cast is nothing more than your way of saying to the compiler, "Trust me. I'm a professional. I take full responsibility for anything weird that happens when those top bits are chopped off."

That brings us to the compound assignment operators. The following will compile,

```
byte b = 3;
b += 7;         // No problem - adds 7 to b (result is 10)
```

and is equivalent to

```
byte b = 3;
b = (byte) (b + 7);  // Won't compile without the
                     // cast, since b + 7 results in an int
```

The compound assignment operator += lets you add to the value of b, without putting in an explicit cast. In fact, +=, -=, *=, and /= will all put in an implicit cast.

Assigning One Primitive Variable to Another Primitive Variable

When you assign one primitive variable to another, the contents of the right-hand variable are copied. For example,

```
int a = 6;
int b = a;
```

This code can be read as, "Assign the bit pattern for the number 6 to the int variable a. Then copy the bit pattern in a, and place the copy into variable b."

So, both variables now hold a bit pattern for 6, but the two variables have no other relationship. We used the variable a *only* to copy its contents. At this point, a and b have identical contents (in other words, identical values), but if we change the contents of *either* a or b, the other variable won't be affected.

Take a look at the following example:

```
class ValueTest {
   public static void main (String [] args) {
      int a = 10;  // Assign a value to a
      System.out.println("a = " + a);
      int b = a;
      b = 30;
      System.out.println("a = " + a + " after change to b");
   }
}
```

The output from this program is

```
%java ValueTest
a = 10
a = 10 after change to b
```

Notice the value of a stayed at 10. The key point to remember is that even after you assign a to b, a and b are not referring to the same place in memory. The a and b variables do not share a single value; they have identical copies.

Reference Variable Assignments

You can assign a newly created object to an object reference variable as follows:

```
Button b = new Button();
```

The preceding line does three key things:

- Makes a reference variable named `b`, of type `Button`
- Creates a new `Button` object on the heap
- Assigns the newly created `Button` object to the reference variable `b`

You can also assign `null` to an object reference variable, which simply means the variable is not referring to any object:

```
Button c = null;
```

The preceding line creates space for the Button reference variable (the bit holder for a reference value), but doesn't create an actual Button object.

As we discussed in the last chapter, you can also use a reference variable to refer to any object that is a subclass of the declared reference variable type, as follows:

```
public class Foo {
   public void doFooStuff() { }
}
public class Bar extends Foo {
   public void doBarStuff() { }
}
class Test {
   public static void main (String [] args) {
      Foo reallyABar = new Bar();  // Legal because Bar is a
                                   // subclass of Foo
      Bar reallyAFoo = new Foo();  // Illegal! Foo is not a
                                   // subclass of Bar
   }
}
```

The rule is that you can assign a subclass of the declared type, but not a superclass of the declared type. Remember, a Bar object is guaranteed to be able to do anything a Foo can do, so anyone with a Foo reference can invoke Foo methods even though the object is actually a Bar.

In the preceding code, we see that Foo has a method `doFooStuff()` that someone with a Foo reference might try to invoke. If the object referenced by the Foo variable is really a Foo, no problem. But it's also no problem if the object is a Bar, since Bar inherited the `doFooStuff()` method. You can't make it work

in reverse, however. If somebody has a Bar reference, they're going to invoke `doBarStuff()`, but if the object is a Foo, it won't know how to respond.

Variable Scope

Once you've declared and initialized a variable, a natural question is "How long will this variable be around?" This is a question regarding the scope of variables. And not only is scope an important thing to understand in general, it also plays a big part in the exam. Let's start by looking at a class file:

```
class Layout {                          // class

  static int s = 343;                   // static variable

  int x;                                // instance variable

  { x = 7; int x2 = 5; }                // initialization block

  Layout() { x += 8; int x3 = 6;}       // constructor

  void doStuff() {                      // method

    int y = 0;                          // local variable

    for(int z = 0; z < 4; z++) {        // 'for' code block
      y += z + x;
    }
  }
}
```

As with variables in all Java programs, the variables in this program (`s`, `x`, `x2`, `x3`, `y`, and `z`) all have a scope:

- `s` is a static variable.
- `x` is an instance variable.
- `y` is a local variable (sometimes called a "method local" variable).
- `z` is a block variable.
- `x2` is an init block variable, a flavor of local variable.
- `x3` is a constructor variable, a flavor of local variable.

For the purposes of discussing the scope of variables, we can say that there are four basic scopes:

- Static variables have the longest scope; they are created when the class is loaded, and they survive as long as the class stays loaded in the JVM.
- Instance variables are the next most long-lived; they are created when a new instance is created, and they live until the instance is removed.
- Local variables are next; they live as long as their method remains on the stack. As we'll soon see, however, local variables can be alive, and still be "out of scope".
- Block variables live only as long as the code block is executing.

Scoping errors come in many sizes and shapes. One common mistake happens when a variable is *shadowed* and two scopes overlap. We'll take a detailed look at shadowing in a few pages. The most common reason for scoping errors is when you attempt to access a variable that is not in scope. Let's look at three common examples of this type of error:

- Attempting to access an instance variable from a static context (typically from `main()`).

```
class ScopeErrors {
  int x = 5;
  public static void main(String[] args) {
    x++;   // won't compile, x is an 'instance' variable
  }
}
```

- Attempting to access a local variable from a nested method.

  When a method, say `go()`, invokes another method, say `go2()`, `go2()` won't have access to `go()`'s local variables. While `go2()` is executing, `go()`'s local variables are still *alive*, but they are *out of scope*. When `go2()` completes, it is removed from the stack, and `go()` resumes execution. At this point, all of `go()`'s previously declared variables are back in scope. For example:

```
class ScopeErrors {
  public static void main(String [] args) {
    ScopeErrors s = new ScopeErrors();
    s.go();
  }
  void go() {
    int y = 5;
```

```
    go2();
    y++;        // once go2() completes, y is back in scope
  }
  void go2() {
    y++;        // won't compile, y is local to go()
  }
}
```

- Attempting to use a block variable after the code block has completed.

  It's very common to declare and use a variable within a code block, but be careful not to try to use the variable once the block has completed:

```
void go3() {
  for(int z = 0; z < 5; z++) {
    boolean test = false;
    if(z == 3) {
      test = true;
      break;
    }
  }
  System.out.print(test);   // 'test' is an ex-variable,
                            // it has ceased to be...
}
```

In the last two examples, the compiler will say something like this:

```
cannot find symbol
```

This is the compiler's way of saying, "That variable you just tried to use? Well, it might have been valid in the distant past (like one line of code ago), but this is Internet time baby, I have no memory of such a variable."

exam Watch

Pay extra attention to code block scoping errors. You might see them in switches, try-catches, for, do, and while loops.

Using a Variable or Array Element That Is Uninitialized and Unassigned

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).

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, although they may be given an explicit value after the object's super-constructors have completed. Table 3-1 lists the default values for primitive and object types.

TABLE 3-1 Default Values for Primitives 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'`

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.

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;          // instance reference variable
  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. The following modification to the Book code runs into trouble:

```
public class Book {
  private String title;          // instance reference variable
  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 something like this:

```
Exception in thread "main" java.lang.NullPointerException
     at Book.main(Book.java:9)
```

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;          // instance reference variable
  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

Later in this chapter we'll be taking a very detailed look at declaring, constructing, and initializing arrays and multidimensional arrays. For now, we're just going to look at the rule for an array element's default values.

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.*

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, and they include a method's 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, or 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);
  }
}
```

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 output something like this:

```
%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).

on the Job

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 compiler will choke here
  }
}
```

The compiler will produce an error something like this:

```
TestLocal.java:9: variable x might not have been initialized
```

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 Object References

Objects references, 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 an error similar to the following:

```
%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.

Assigning One Reference Variable to Another

With primitive variables, an assignment of one variable to another means the contents (bit pattern) of one variable are *copied* into another. Object reference variables work exactly the same way. The contents of a reference variable are a bit pattern, so if you assign reference variable `a` to reference variable `b`, the bit pattern in `a` is *copied* and the new *copy* is placed into `b`. (Some people have created a game around counting how many times we use the word *copy* in this chapter…this copy concept is a biggie!) If we assign an existing instance of an object to a new reference variable, then two reference variables will hold the same bit pattern—a bit pattern referring to a specific object on the heap. Look at the following code:

```
import java.awt.Dimension;
class ReferenceTest {
  public static void main (String [] args) {
    Dimension a = new Dimension(5,10);
    System.out.println("a.height = " + a.height);
    Dimension b = a;
    b.height = 30;
    System.out.println("a.height = " + a.height +
                       " after change to b");
  }
}
```

In the preceding example, a Dimension object `a` is declared and initialized with a width of 5 and a height of 10. Next, Dimension `b` is declared, and assigned the value of `a`. At this point, both variables (`a` and `b`) hold identical values, because the contents of a were copied into `b`. There is still only one Dimension object—the one that both `a` and `b` refer to. Finally, the height property is changed using the `b` reference. Now think for a minute: is this going to change the height property of `a` as well? Let's see what the output will be:

```
%java ReferenceTest
a.height = 10
a.height = 30 after change to b
```

From this output, we can conclude that both variables refer to the same instance of the Dimension object. When we made a change to `b`, the height property was also changed for `a`.

One exception to the way object references are assigned is String. In Java, String objects are given special treatment. For one thing, String objects are immutable; you can't change the value of a String object (lots more on this concept in Chapter 6). But it sure looks as though you can. Examine the following code:

```
class StringTest {
  public static void main(String [] args) {
    String x = "Java";  // Assign a value to x
    String y = x;       // Now y and x refer to the same
                        // String object

    System.out.println("y string = " + y);
    x = x + " Bean";    // Now modify the object using
                        // the x reference
```

```
    System.out.println("y string = " + y);
  }
}
```

You might think String `y` will contain the characters `Java Bean` after the variable `x` is changed, because Strings are objects. Let's see what the output is:

```
%java StringTest
y string = Java
y string = Java
```

As you can see, even though `y` is a reference variable to the same object that `x` refers to, when we change `x`, it doesn't change `y`! For any other object type, where two references refer to the same object, if either reference is used to modify the object, both references will see the change because there is still only a single object. *But any time we make any changes at all to a String, the VM will update the reference variable to refer to a different object*. The different object might be a new object, or it might not, but it will definitely be a different object. The reason we can't say for sure whether a new object is created is because of the String constant pool, which we'll cover in Chapter 6.

You need to understand what happens when you use a String reference variable to modify a string:

- A new string is created (or a matching String is found in the String pool), leaving the original String object untouched.
- The reference used to modify the String (or rather, make a new String by modifying a copy of the original) is then assigned the brand new String object.

So when you say

```
1. String s = "Fred";
2. String t = s;     // Now t and s refer to the same
                     // String object
3. t.toUpperCase();  // Invoke a String method that changes
                     // the String
```

you haven't changed the original String object created on line 1. When line 2 completes, both `t` and `s` reference the same String object. But when line 3 runs, rather than modifying the object referred to by `t` (which is the one and only String

object up to this point), a brand new String object is created. And then abandoned. Because the new String isn't assigned to a String variable, the newly created String (which holds the string "FRED") is toast. So while two String objects were created in the preceding code, only one is actually referenced, and both `t` and `s` refer to it. The behavior of Strings is extremely important in the exam, so we'll cover it in much more detail in Chapter 6.

CERTIFICATION OBJECTIVE

Passing Variables into Methods (Exam Objective 7.3)

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.

Methods can be declared to take primitives and/or object references. You need to know how (or if) the caller's variable can be affected by the called method. The difference between object reference and primitive variables, when passed into methods, is huge and important. To understand this section, you'll need to be comfortable with the assignments section covered in the first part of this chapter.

Passing Object Reference Variables

When you pass an object variable into a method, you must keep in mind that you're passing the object *reference*, and not the actual object itself. Remember that a reference variable holds bits that represent (to the underlying VM) a way to get to a specific object in memory (on the heap). More importantly, you must remember that you aren't even passing the actual reference variable, but rather a *copy* of the reference variable. A copy of a variable means you get a copy of the bits in that variable, so when you pass a reference variable, you're passing a copy of the bits representing how to get to a specific object. In other words, both the caller and the called method will now have identical copies of the reference, and thus both will refer to the same exact (*not* a copy) object on the heap.

For this example, we'll use the Dimension class from the java.awt package:

```
1. import java.awt.Dimension;
2. class ReferenceTest {
```

```
 3.   public static void main (String [] args) {
 4.     Dimension d = new Dimension(5,10);
 5.     ReferenceTest rt = new ReferenceTest();
 6.     System.out.println("Before modify() d.height = "
                           + d.height);
 7.     rt.modify(d);
 8.     System.out.println("After modify() d.height = "
                           + d.height);
 9.   }
10.   void modify(Dimension dim) {
11.     dim.height = dim.height + 1;
12.     System.out.println("dim = " + dim.height);
13.   }
14. }
```

When we run this class, we can see that the `modify()` method was indeed able to modify the original (and only) Dimension object created on line 4.

```
C:\Java Projects\Reference>java ReferenceTest
Before modify() d.height = 10
dim = 11
After modify() d.height = 11
```

Notice when the Dimension object on line 4 is passed to the `modify()` method, any changes to the object that occur inside the method are being made to the object whose reference was passed. In the preceding example, reference variables `d` and `dim` both point to the same object.

Does Java Use Pass-By-Value Semantics?

If Java passes objects by passing the reference variable instead, does that mean Java uses pass-by-reference for objects? Not exactly, although you'll often hear and read that it does. Java is actually pass-by-value for all variables running within a single VM. Pass-by-value means pass-by-variable-value. And that means, pass-by-copy-of-the-variable! (There's that word *copy* again!)

It makes no difference if you're passing primitive or reference variables, you are always passing a copy of the bits in the variable. So for a primitive variable, you're passing a copy of the bits representing the value. For example, if you pass an `int` variable with the value of `3`, you're passing a copy of the bits representing `3`. The called method then gets its own copy of the value, to do with it what it likes.

And if you're passing an object reference variable, you're passing a copy of the bits representing the reference to an object. The called method then gets its own copy of the reference variable, to do with it what it likes. But because two identical reference variables refer to the exact same object, if the called method modifies the object (by invoking setter methods, for example), the caller will see that the object the caller's original variable refers to has also been changed. In the next section, we'll look at how the picture changes when we're talking about primitives.

The bottom line on pass-by-value: the called method can't change the caller's variable, although for object reference variables, the called method can change the object the variable referred to. What's the difference between changing the variable and changing the object? For object references, it means the called method can't reassign the caller's original reference variable and make it refer to a different object, or `null`. For example, in the following code fragment,

```
void bar() {
   Foo f = new Foo();
   doStuff(f);
}
void doStuff(Foo g) {
   g.setName("Boo");
   g = new Foo();
}
```

reassigning `g` does not reassign `f`! At the end of the `bar()` method, two Foo objects have been created, one referenced by the local variable `f` and one referenced by the local (argument) variable `g`. Because the `doStuff()` method has a copy of the reference variable, it has a way to get to the original Foo object, for instance to call the `setName()` method. But, the `doStuff()` method does *not* have a way to get to the `f` reference variable. So `doStuff()` can change values within the object `f` refers to, but `doStuff()` can't change the actual contents (bit pattern) of `f`. In other words, `doStuff()` can change the state of the object that `f` refers to, but it can't make `f` refer to a different object!

Passing Primitive Variables

Let's look at what happens when a primitive variable is passed to a method:

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

```
    int a = 1;
    ReferenceTest rt = new ReferenceTest();
    System.out.println("Before modify() a = " + a);
    rt.modify(a);
    System.out.println("After modify() a = " + a);
  }
  void modify(int number) {
    number = number + 1;
    System.out.println("number = " + number);
  }
}
```

In this simple program, the variable `a` is passed to a method called `modify()`, which increments the variable by 1. The resulting output looks like this:

```
Before modify() a = 1
number = 2
After modify() a = 1
```

Notice that `a` did not change after it was passed to the method. Remember, it was a copy of `a` that was passed to the method. When a primitive variable is passed to a method, it is passed by value, which means pass-by-copy-of-the-bits-in-the-variable.

FROM THE CLASSROOM

The Shadowy World of Variables

Just when you think you've got it all figured out, you see a piece of code with variables not behaving the way you think they should. You might have stumbled into code with a shadowed variable. You can shadow a variable in several ways. We'll look at the one most likely to trip you up: hiding an instance variable by shadowing it with a local variable. Shadowing involves redeclaring a variable that's already been declared somewhere else.

The effect of shadowing is to hide the previously declared variable in such a way that it may look as though you're using the hidden variable, but you're actually using the shadowing variable. You might find reasons to shadow a variable intentionally, but typically it happens by accident and causes hard-to-find bugs. On the exam, you can expect to see questions where shadowing plays a role.

FROM THE CLASSROOM

You can shadow an instance variable by declaring a local variable of the same name, either directly or as part of an argument:

```
class Foo {
  static int size = 7;
  static void changeIt(int size) {
    size = size + 200;
    System.out.println("size in changeIt is " + size);
  }
  public static void main (String [] args) {
    Foo f = new Foo();
    System.out.println("size = " + size);
    changeIt(size);
    System.out.println("size after changeIt is " + size);
  }
}
```

The preceding code appears to change the `size` instance variable in the `changeIt()` method, but because `changeIt()` has a parameter named `size`, the local `size` variable is modified while the instance variable `size` is untouched. Running class Foo prints

```
%java Foo
size = 7
size in changeIt is 207
size after changeIt is 7
```

Things become more interesting when the shadowed variable is an object reference, rather than a primitive:

```
class Bar {
  int barNum = 28;
}
```

FROM THE CLASSROOM

```
class Foo {
  Bar myBar = new Bar();
  void changeIt(Bar myBar) {
    myBar.barNum = 99;
    System.out.println("myBar.barNum in changeIt is " + myBar.barNum);
    myBar = new Bar();
    myBar.barNum = 420;
    System.out.println("myBar.barNum in changeIt is now " + myBar.barNum);
  }
  public static void main (String [] args) {
    Foo f = new Foo();
    System.out.println("f.myBar.barNum is " + f.myBar.barNum);
    f.changeIt(f.myBar);
    System.out.println("f.myBar.barNum after changeIt is "
                       + f.myBar.barNum);
  }
}
```

The preceding code prints out this:

```
f.myBar.barNum is 28
myBar.barNum in changeIt is 99
myBar.barNum in changeIt is now 420
f.myBar.barNum after changeIt is 99
```

You can see that the shadowing variable (the local parameter `myBar` in `changeIt()`) can still affect the `myBar` instance variable, because the `myBar` parameter receives a reference to the same Bar object. But when the local `myBar` is reassigned a new Bar object, which we then modify by changing its `barNum` value, Foo's original `myBar` instance variable is untouched.

CERTIFICATION OBJECTIVE

Array Declaration, Construction, and Initialization (Exam Objective 1.3)

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.

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, and so on) 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 7 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;  // brackets before name (recommended)
int key []; // brackets after name (legal but less readable)
            // spaces between the name and [] legal, but bad
```

Declaring an array of object references:

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

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;  // recommended
String[] ManagerName [];    // yucky, but legal
```

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.

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* (where all objects live)—i.e., doing a `new` on the array type. To create an array object, Java must know

how much space to allocate on the heap, so you must specify the size of the array at creation time. The size of the array is the number of elements the array will hold.

Constructing One-Dimensional Arrays

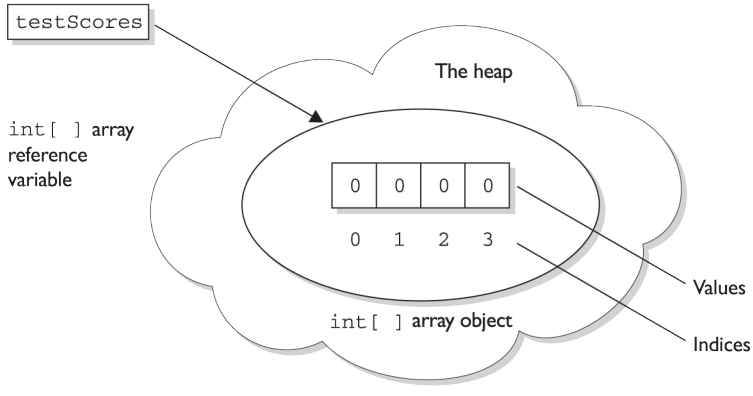
The most straightforward way to construct an array is to use the keyword new followed by the array type, with a bracket specifying how many elements of that type the array will hold. The following is an example of constructing an array of type int:

```
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 that will hold four `int`s, and assign it to the reference variable named `testScores`. Also, go ahead and set each int element to zero. Thanks." (The compiler appreciates good manners.) Figure 3-2 shows the `testScores` array on the heap, after construction.

FIGURE 3-2

A one-dimensional array on the Heap



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

```
int[] testScores = new int[4];
```

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];
```

Remember 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 statement, 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 on the heap." This also implies that the object's constructor runs, as a result of the construct/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 the next section. 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`:

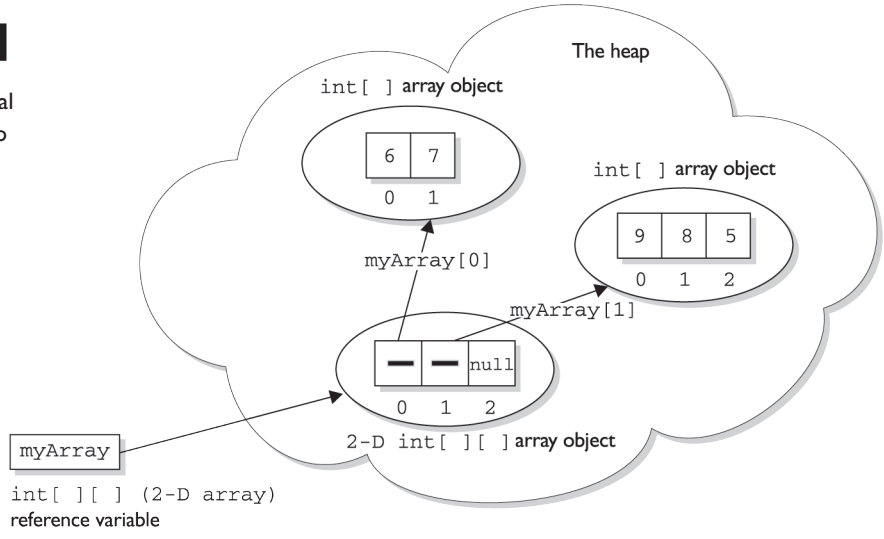
```
int[][] myArray = new int[3][];
```

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 `myArray`.

Figure 3-3 shows how a two-dimensional `int` array works on the heap.

FIGURE 3-3

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;
```

Initializing an Array

Initializing an array means putting things into it. The "things" in the array are the array's elements, and they're either primitive values (2, `x`, `false`, and so on), 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];
```

We have one array object on the heap, with three `null` references of type Animal, but we don't 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.

exam Watch

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!

exam Watch

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 elements

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. As we'll cover in Chapter 5, as of Java 5, we could have written the `for` loop without using the `length` variable:

```
for(Dog d : myDogs)
  d = new Dog();
```

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 = {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 three (three elements).
- Populates the array's elements with the values 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 comma-separated items between the curly braces. The code is functionally equivalent to the following longer code:

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

This begs the question, "Why would anyone use the longer way?" One reason come to mind. You might not know—at the time you create the array—the values that will be assigned to the array's elements. This array shortcut alone (combined with the stimulating prose) is worth the price of this book.

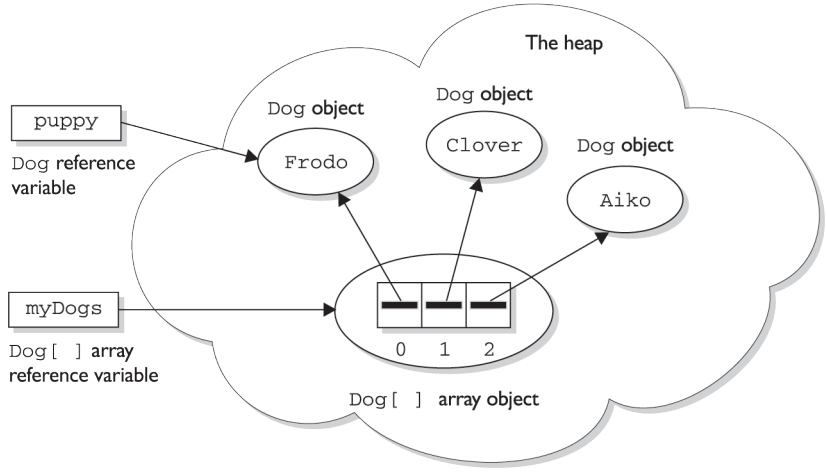
With object references rather than primitives, it works exactly the same way:

```
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. It also creates two new Dog objects (`Clover` and `Aiko`), and adds them to the last two Dog reference variable elements in the myDogs array. Figure 3-4 shows the result.

FIGURE 3-4

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` and by `myDogs[0]`
1 `Dog[ ]` array referenced by `myDogs`
2 `Dog` objects referenced by `myDogs[1]` and `myDogs[2]`

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 2
scores[2][1] // the int value 4
```

Figure 3-5 shows the result of declaring, constructing, and initializing a two-dimensional array in one statement.

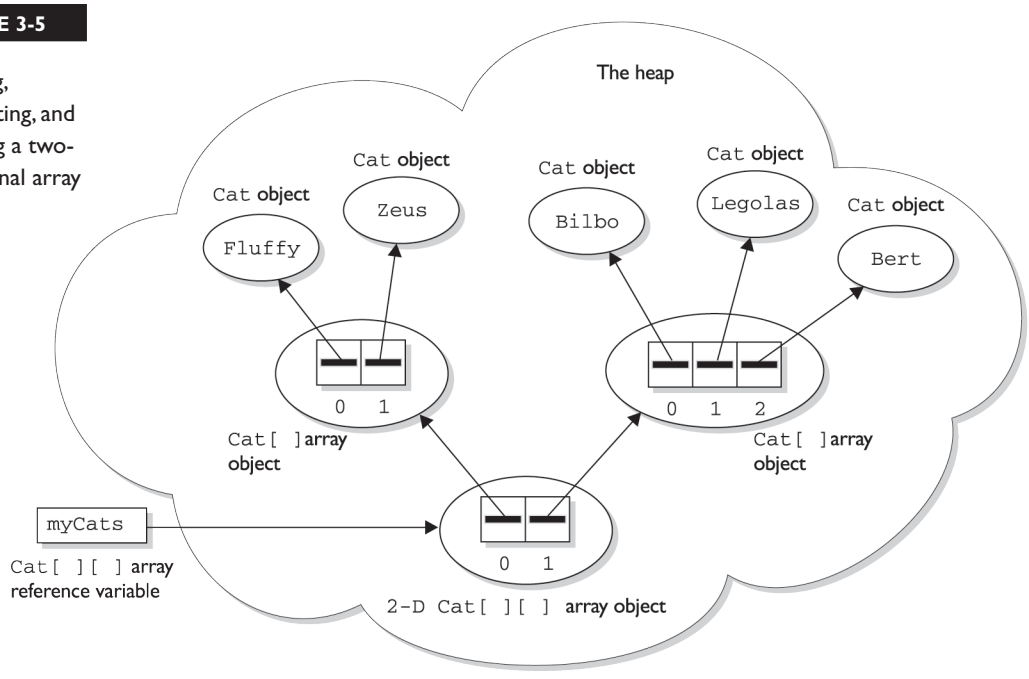
Constructing and Initializing an Anonymous Array

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};
```

FIGURE 3-5

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

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. 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 answers 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. For 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 Subaru is a subclass of Car, you can put both Subaru objects and Car objects into an array of type Car as follows:

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

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 (plus it doesn't make 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 assigned 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 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;     // NO, expecting an int array not an int
books[0] = numbers;     // OK, numbers is an int array
```

Figure 3-6 shows an example of legal and illegal assignments for references to an array.

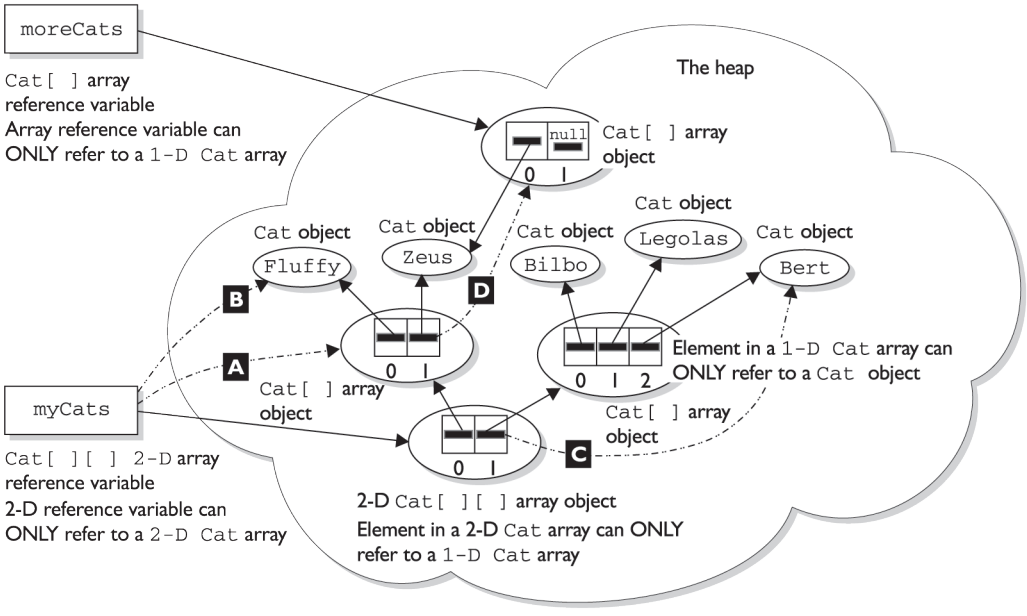
Initialization Blocks

We've talked about two places in a class where you can put code that performs operations: methods and constructors. Initialization blocks are the third place in a Java program where operations can be performed. Initialization blocks run when the class is first loaded (a static initialization block) or when an instance is created (an instance initialization block). Let's look at an example:

```
class SmallInit {
    static int x;
    int y;

    static { x = 7 ; }       // static init block
    { y = 8; }               // instance init block
  }
```

FIGURE 3-6 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

Illegal

As you can see, the syntax for initialization blocks is pretty terse. They don't have names, they can't take arguments, and they don't return anything. A *static* initialization block runs *once*, when the class is first loaded. An *instance* initialization block runs once *every time a new instance is created*. Remember when we talked about the order in which constructor code executed? Instance init block code runs right

after the call to `super()` in a constructor, in other words, after all super-constructors have run.

You can have many initialization blocks in a class. It is important to note that unlike methods or constructors, *the order in which initialization blocks appear in a class matters*. When it's time for initialization blocks to run, if a class has more than one, they will run in the order in which they appear in the class file…in other words, from the top down. Based on the rules we just discussed, can you determine the output of the following program?

```
class Init {
  Init(int x) { System.out.println("1-arg const"); }
  Init() { System.out.println("no-arg const"); }
  static { System.out.println("1st static init"); }
  { System.out.println("1st instance init"); }
  { System.out.println("2nd instance init"); }
  static { System.out.println("2nd static init"); }

  public static void main(String [] args) {
    new Init();
    new Init(7);
  }
}
```

To figure this out, remember these rules:

- Init blocks execute in the order they appear.
- Static init blocks run once, when the class is first loaded.
- Instance init blocks run every time a class instance is created.
- Instance init blocks run after the constructor's call to `super()`.

With those rules in mind, the following output should make sense:

```
1st static init
2nd static init
1st instance init
2nd instance init
no-arg const
1st instance init
2nd instance init
1-arg const
```

As you can see, the instance init blocks each ran twice. Instance init blocks are often used as a place to put code that all the constructors in a class should share. That way, the code doesn't have to be duplicated across constructors.

Finally, if you make a mistake in your static init block, the JVM can throw an `ExceptionInInitializationError`. Let's look at an example,

```
class InitError {
  static int [] x = new int[4];
  static { x[4] = 5; }           // bad array index!

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

which produces something like:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: 4
        at InitError.<clinit>(InitError.java:3)
```

exam Watch

By convention, init blocks usually appear near the top of the class file, somewhere around the constructors. However, this is the SCJP exam we're talking about. Don't be surprised if you find an init block tucked in between a couple of methods, looking for all the world like a compiler error waiting to happen!

CERTIFICATION OBJECTIVE

Using Wrapper Classes and Boxing (Exam Objective 3.1)

3.1 Develop code that uses the primitive wrapper classes (such as Boolean, Character, Double, Integer, etc.), and/or autoboxing & unboxing. Discuss the differences between the String, StringBuilder, and StringBuffer classes.

The wrapper classes in the Java API serve two primary purposes:

- To provide a mechanism to "wrap" primitive values in an object so that the primitives can be included in activities reserved for objects, like being added to Collections, or returned from a method with an object return value. Note: With Java 5's addition of autoboxing (and unboxing), which we'll get to in a few pages, many of the wrapping operations that programmers used to do manually are now handled automatically.
- To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

An Overview of the Wrapper Classes

There is a wrapper class for every primitive in Java. For instance, the wrapper class for `int` is Integer, the class for `float` is Float, and so on. Remember that the primitive name is simply the lowercase name of the wrapper except for `char`, which maps to Character, and `int`, which maps to Integer. Table 3-2 lists the wrapper classes in the Java API.

TABLE 3-2 Wrapper Classes and Their Constructor Arguments

Primitive	Wrapper Class	Constructor Arguments
`boolean`	`Boolean`	`boolean` or `String`
`byte`	`Byte`	`byte` or `String`
`char`	`Character`	`char`
`double`	`Double`	`double` or `String`
`float`	`Float`	`float`, `double`, or `String`
`int`	`Integer`	`int` or `String`
`long`	`Long`	`long` or `String`
`short`	`Short`	`short` or `String`

Creating Wrapper Objects

For the exam you need to understand the three most common approaches for creating wrapper objects. Some approaches take a String representation of a primitive as an argument. Those that take a String throw NumberFormatException if the String provided cannot be parsed into the appropriate primitive. For example "two" can't be parsed into "2". *Wrapper objects are immutable***.** Once they have been given a value, that value cannot be changed. We'll talk more about wrapper immutability when we discuss boxing in a few pages.

The Wrapper Constructors

All of the wrapper classes except Character provide two constructors: one that takes a primitive of the type being constructed, and one that takes a String representation of the type being constructed—for example,

```
Integer i1 = new Integer(42);
Integer i2 = new Integer("42");
```

or

```
Float f1 = new Float(3.14f);
Float f2 = new Float("3.14f");
```

The Character class provides only one constructor, which takes a char as an argument—for example,

```
Character c1 = new Character('c');
```

The constructors for the Boolean wrapper take either a `boolean` value `true` or `false`, or a case-insensitive String with the value "true" or "false". Until Java 5, a Boolean object couldn't be used as an expression in a boolean test—for instance,

```
Boolean b = new Boolean("false");
if (b)      // won't compile, using Java 1.4 or earlier
```

As of Java 5, a Boolean object *can* be used in a boolean test, because the compiler will automatically "un-box" the Boolean to a `boolean`. We'll be focusing on Java 5's autoboxing capabilities in the very next section—so stay tuned!

The valueOf() Methods

The two (well, usually two) static `valueOf()` methods provided in most of the wrapper classes give you another approach to creating wrapper objects. Both methods take a String representation of the appropriate type of primitive as their first argument, the second method (when provided) takes an additional argument, `int radix`, which indicates in what base (for example binary, octal, or hexadecimal) the first argument is represented—for example,

```
Integer i2 = Integer.valueOf("101011", 2);  // converts 101011
                                            // to 43 and
                                            // assigns the value
                                            // 43 to the
                                            // Integer object i2
```

or

```
Float f2 = Float.valueOf("3.14f");   // assigns 3.14 to the
                                     // Float object f2
```

Using Wrapper Conversion Utilities

As we said earlier, a wrapper's second big function is converting stuff. The following methods are the most commonly used, and are the ones you're most likely to see on the test.

xxxValue()

When you need to convert the value of a wrapped numeric to a primitive, use one of the many `xxxValue()` methods. All of the methods in this family are no-arg methods. As you can see by referring to Table 3-3, there are 36 `xxxValue()` methods. Each of the six numeric wrapper classes has six methods, so that any numeric wrapper can be converted to any primitive numeric type—for example,

```
Integer i2 = new Integer(42);  //  make a new wrapper object
byte b = i2.byteValue();       //  convert i2's value to a byte
                               //  primitive
short s = i2.shortValue();     //  another of Integer's xxxValue
                               //  methods
double d = i2.doubleValue();   //  yet another of Integer's
                               //  xxxValue methods
```

or

```
Float f2 = new Float(3.14f);   // make a new wrapper object
short s = f2.shortValue();     // convert f2's value to a short
                               // primitive
System.out.println(s);         // result is 3  (truncated, not
                               // rounded)
```

parseXxx() and valueOf()

The six `parseXxx()` methods (one for each numeric wrapper type) are closely related to the `valueOf()` method that exists in all of the numeric wrapper classes. Both `parseXxx()` and `valueOf()` take a String as an argument, throw a NumberFormatException (a.k.a. NFE) if the String argument is not properly formed, and can convert String objects from different bases (radix), when the underlying primitive type is any of the four integer types. (See Table 3-3.) The difference between the two methods is

- `parseXxx()` returns the named primitive.
- `valueOf()` returns a newly created wrapped object of the type that invoked the method.

Here are some examples of these methods in action:

```
double d4 = Double.parseDouble("3.14");   // convert a String
                                          // to a primitive
System.out.println("d4 = " + d4);         // result is d4 = 3.14

Double d5 = Double.valueOf("3.14");       // create a Double obj
System.out.println(d5 instanceof Double); // result is "true"
```

The next examples involve using the radix argument (in this case binary):

```
long L2 = Long.parseLong("101010", 2);    // binary String to a
                                          // primitive
System.out.println("L2 = " + L2);         // result is: L2 = 42

Long L3 = Long.valueOf("101010", 2);      // binary String to
                                          // Long object
System.out.println("L3 value = " + L3);   // result is:
                                          // L3 value = 42
```

toString()

Class Object, the alpha class, has a `toString()` method. Since we know that all other Java classes inherit from class Object, we also know that all other Java classes have a `toString()` method. The idea of the `toString()` method is to allow you to get some meaningful representation of a given object. For instance, if you have a Collection of various types of objects, you can loop through the Collection and print out some sort of meaningful representation of each object using the `toString()` method, which is guaranteed to be in every class. We'll talk more about `toString()` in the Collections chapter, but for now let's focus on how `toString()` relates to the wrapper classes which, as we know, are marked `final`. All of the wrapper classes have a no-arg, nonstatic, instance version of `toString()`. This method returns a String with the value of the primitive wrapped in the object—for instance,

```
Double d = new Double("3.14");
System.out.println("d = "+ d.toString() ); // result is d = 3.14
```

All of the numeric wrapper classes provide an overloaded, `static toString()` method that takes a primitive numeric of the appropriate type (`Double.toString()` takes a `double`, `Long.toString()` takes a `long`, and so on) and, of course, returns a String:

```
String d = Double.toString(3.14);      // d = "3.14"
```

Finally, Integer and Long provide a third toString() method. It's static, its first argument is the primitive, and its second argument is a radix. The radix tells the method to take the first argument, which is radix 10 (base 10) by default, and convert it to the radix provided, then return the result as a String—for instance,

```
String s = "hex = "+ Long.toString(254,16); // s = "hex = fe"
```

toXxxString() (Binary, Hexadecimal, Octal)

The Integer and Long wrapper classes let you convert numbers in base 10 to other bases. These conversion methods, `toXxxString()`, take an `int` or `long`, and return a String representation of the converted number, for example,

```
String s3 = Integer.toHexString(254);  // convert 254 to hex
System.out.println("254 is " + s3);    // result: "254 is fe"

String s4 = Long.toOctalString(254); // convert 254 to octal
System.out.print("254(oct) ="+ s4);  // result: "254(oct) =376"
```

Studying Table 3-3 is the single best way to prepare for this section of the test. If you can keep the differences between `xxxValue()`, `parseXxx()`, and `valueOf()` straight, you should do well on this part of the exam.

TABLE 3-3 Common Wrapper Conversion Methods

Method s = static n = NFE exception	**Boolean**	**Byte**	**Character**	**Double**	**Float**	**Integer**	**Long**	**Short**
byteValue		x		x	x	x	x	x
doubleValue		x		x	x	x	x	x
floatValue		x		x	x	x	x	x
intValue		x		x	x	x	x	x
longValue		x		x	x	x	x	x
shortValue		x		x	x	x	x	x
parseXxx s,n		x		x	x	x	x	x
parseXxx s,n (with radix)		x				x	x	x
valueOf s,n	x	x		x	x	x	x	x
valueOf s,n (with radix)		x				x	x	x
toString	x	x	x	x	x	x	x	x
toString s (primitive)	x	x	x	x	x	x	x	x
toString s (primitive, radix)						x	x	

In summary, the essential method signatures for Wrapper conversion methods are

primitive xxxValue() - to convert a Wrapper to a primitive

primitive parseXxx(String) - to convert a String to a primitive

Wrapper valueOf(String) - to convert a String to a Wrapper

Autoboxing

New to Java 5 is a feature known variously as: autoboxing, auto-unboxing, boxing, and unboxing. We'll stick with the terms boxing and unboxing. Boxing and unboxing make using wrapper classes more convenient. In the old, pre-Java 5 days, if you wanted to make a wrapper, unwrap it, use it, and then rewrap it, you might do something like this:

```
Integer y = new Integer(567);    // make it
int x = y.intValue();            // unwrap it
x++;                             // use it
y = new Integer(x);              // re-wrap it
System.out.println("y = " + i);  // print it
```

Now, with new and improved Java 5 you can say

```
Integer y = new Integer(567);    // make it
y++;                             // unwrap it, increment it,
                                 // rewrap it
System.out.println("y = " + i);  // print it
```

Both examples produce the output:

```
y = 568
```

And yes, you read that correctly. The code appears to be using the post-increment operator on an object reference variable! But it's simply a convenience. Behind the scenes, the compiler does the unboxing and reassignment for you. Earlier we mentioned that wrapper objects are immutable... this example appears to contradict that statement. It sure looks like y's value changed from 567 to 568. What actually happened, is that a second wrapper object was created and its value was set to 568. If only we could access that first wrapper object, we could prove it...

Let's try this:

```
Integer y = 567;                 // make a wrapper
Integer x = y;                   // assign a second ref
                                 // var to THE wrapper

System.out.println(y==x);        // verify that they refer
                                 // to the same object
```

```
y++;                                  // unwrap, use, "rewrap"
System.out.println(x + " " + y);      // print values

System.out.println(y==x);             // verify that they refer
                                      // to different objects
```

Which produces the output:

```
true
567 568
false
```

So, under the covers, when the compiler got to the line i++; it had to substitute something like this:

```
int x2 = y.intValue();            // unwrap it
x2++;                             // use it
y = new Integer(x2);              // re-wrap it
```

Just as we suspected, there's gotta be a call to `new` in there somewhere.

Boxing, ==, and equals()

We just used == to do a little exploration of wrappers. Let's take a more thorough look at how wrappers work with ==, !=, and `equals()`. We'll talk a lot more about the `equals()` method in later chapters. For now all we have to know is that the intention of the `equals()` method is to determine whether two instances of a given class are "meaningfully equivalent." This definition is intentionally subjective; it's up to the creator of the class to determine what "equivalent" means for objects of the class in question. The API developers decided that for all the wrapper classes, two objects are equal if they are of the same type and have the same value. It shouldn't be surprising that

```
Integer i1 = 1000;
Integer i2 = 1000;
if(i1 != i2) System.out.println("different objects");
if(i1.equals(i2)) System.out.println("meaningfully equal");
```

Produces the output:

```
different objects
meaningfully equal
```

It's just two wrapper objects that happen to have the same value. Because they have the same `int` value, the `equals()` method considers them to be "meaningfully equivalent", and therefore returns `true`. How about this one:

```
Integer i3 = 10;
Integer i4 = 10;
if(i3 == i4) System.out.println("same object");
if(i3.equals(i4)) System.out.println("meaningfully equal");
```

This example produces the output:

```
same object
meaningfully equal
```

Yikes! The `equals()` method seems to be working, but what happened with == and != ? Why is != telling us that `i1` and `i2` are different objects, when == is saying that `i3` and `i4` are the same object? In order to save memory, two instances of the following wrapper objects will always be == when their primitive values are the same:

- `Boolean`
- `Byte`
- `Character` from \u0000 to \u007f (7f is 127 in decimal)
- `Short` and `Integer` from -128 to 127

Where Boxing Can Be Used

As we discussed earlier, it's very common to use wrappers in conjunction with collections. Any time you want your collection to hold objects and primitives, you'll want to use wrappers to make those primitives collection-compatible. The general rule is that boxing and unboxing work wherever you can normally use a primitive or a wrapped object. The following code demonstrates some legal ways to use boxing:

```
class UseBoxing {
  public static void main(String [] args) {
    UseBoxing u = new UseBoxing();
    u.go(5);
  }

  boolean go(Integer i) {       // boxes the int it was passed
    Boolean ifSo = true;        // boxes the literal
    Short s = 300;              // boxes the primitive
```

```
    if(ifSo) {                    // unboxing
      System.out.println(++s);    // unboxes, increments, reboxes
    }
    return !ifSo;                 // unboxes, returns the inverse
  }
}
```

exam Watch

Remember, wrapper reference variables can be null. That means that you have to watch out for code that appears to be doing safe primitive operations, but that could throw a `NullPointerException`***:***

```
class Boxing2 {
  static Integer x;
  public static void main(String [] args) {
    doStuff(x);
  }
  static void doStuff(int z) {
    int z2 = 5;
    System.out.println(z2 + z);
  }
}
```

This code compiles fine, but the JVM throws a `NullPointerException` ***when it attempts to invoke*** `doStuff(x)`***, because*** `x` ***doesn't refer to an*** `Integer` ***object, so there's no value to unbox.***

CERTIFICATION OBJECTIVE

Overloading (Exam Objectives 1.5 and 5.4)

1.5 Given a code example, determine if a method is correctly overriding or overloading another method, and identify legal return values (including covariant returns), for the method.

5.4 Given a scenario, develop code that declares and/or invokes overridden or overloaded methods…

Overloading Made Hard—Method Matching

Although we covered some rules for overloading methods in Chapter 2, in this chapter we've added some new tools to our Java toolkit. In this section we're going to take a look at three factors that can make overloading a little tricky:

- Widening
- Autoboxing
- Var-args

When a class has overloaded methods, one of the compiler's jobs is to determine which method to use whenever it finds an invocation for the overloaded method. Let's look at an example that doesn't use any new Java 5 features:

```
class EasyOver {
  static void go(int x) { System.out.print("int "); }
  static void go(long x) { System.out.print("long "); }
  static void go(double x) { System.out.print("double "); }

  public static void main(String [] args) {
    byte b = 5;
    short s = 5;
    long l = 5;
    float f = 5.0f;

    go(b);
    go(s);
    go(l);
    go(f);
  }
}
```

Which produces the output:

```
int int long double
```

This probably isn't much of a surprise; the calls that use `byte` and the `short` arguments are implicitly widened to match the version of the `go()` method that takes an `int`. Of course, the call with the `long` uses the long version of `go()`, and finally, the call that uses a `float` is matched to the method that takes a `double`.

In every case, when an exact match isn't found, the JVM uses the method with the smallest argument that is wider than the parameter.

You can verify for yourself that if there is only one version of the `go()` method, and it takes a `double`, it will be used to match all four invocations of `go()`.

Overloading with Boxing and Var-args

Now let's take our last example, and add *boxing* into the mix:

```
class AddBoxing {
  static void go(Integer x) { System.out.println("Integer"); }
  static void go(long x) { System.out.println("long"); }

  public static void main(String [] args) {
    int i = 5;
    go(i);           // which go() will be invoked?
  }
}
```

As we've seen earlier, if the only version of the `go()` method was one that took an Integer, then Java 5's boxing capability would allow the invocation of `go()` to succeed. Likewise, if only the `long` version existed, the compiler would use it to handle the `go()` invocation. The question is, given that both methods exist, which one will be used? In other words, does the compiler think that widening a primitive parameter is more desirable than performing an autoboxing operation? The answer is that the compiler will choose widening over boxing, so the output will be

```
long
```

Java 5's designers decided that the most important rule should be that pre-existing code should function the way it used to, so since widening capability already existed, a method that is invoked via widening shouldn't lose out to a newly created method that relies on boxing. Based on that rule, try to predict the output of the following:

```
class AddVarargs {
  static void go(int x, int y) { System.out.println("int,int");}
  static void go(byte... x) { System.out.println("byte... "); }
  public static void main(String[] args) {
    byte b = 5;
    go(b,b);         // which go() will be invoked?
  }
}
```

As you probably guessed, the output is

```
int,int
```

Because, once again, even though each invocation will require some sort of conversion, the compiler will choose the older style before it chooses the newer style, keeping existing code more robust. So far we've seen that

- Widening beats boxing
- Widening beats var-args

At this point, inquiring minds want to know, does boxing beat var-args?

```
class BoxOrVararg {
  static void go(Byte x, Byte y)
                { System.out.println("Byte, Byte"); }
  static void go(byte... x) { System.out.println("byte... "); }

  public static void main(String [] args) {
    byte b = 5;
    go(b,b);          // which go() will be invoked?
  }
}
```

As it turns out, the output is

```
Byte, Byte
```

A good way to remember this rule is to notice that the var-args method is "looser" than the other method, in that it could handle invocations with any number of `int` parameters. A var-args method is more like a catch-all method, in terms of what invocations it can handle, and as we'll see in Chapter 5, it makes most sense for catch-all capabilities to be used as a *last resort*.

Widening Reference Variables

We've seen that it's legal to widen a primitive. Can you widen a reference variable, and if so, what would it mean? Let's think back to our favorite polymorphic assignment:

```
Animal a = new Dog();
```

Along the same lines, an invocation might be:

```
class Animal {static void eat() { } }

class Dog3 extends Animal {
  public static void main(String[] args) {
    Dog3 d = new Dog3();
    d.go(d);              // is this legal ?
  }
  void go(Animal a) { }
}
```

No problem! The `go()` method needs an Animal, and `Dog3` IS-A Animal. (Remember, the `go()` method thinks it's getting an Animal object, so it will only ask it to do Animal things, which of course anything that inherits from Animal can do.) So, in this case, the compiler widens the `Dog3` reference to an Animal, and the invocation succeeds. The key point here is that reference widening depends on inheritance, in other words the IS-A test. Because of this, it's not legal to widen from one wrapper class to another, because the wrapper classes are peers to one another. For instance, it's NOT valid to say that Short IS-A Integer.

exam Watch

It's tempting to think that you might be able to widen an Integer wrapper to a Long wrapper, but the following will NOT compile:

```
class Dog4 {
  public static void main(String [] args) {
    Dog4 d = new Dog4();
    d.test(new Integer(5));  // can't widen an Integer
                             // to a Long
  }
  void test(Long x) { }
}
```

Remember, none of the wrapper classes will widen from one to another! Bytes won't widen to Shorts, Shorts won't widen to Longs, etc.

Overloading When Combining Widening and Boxing

We've looked at the rules that apply when the compiler can match an invocation to a method by performing a single conversion. Now let's take a look at what happens when more than one conversion is required. In this case the compiler will have to widen and then autobox the parameter for a match to be made:

```
class WidenAndBox {
  static void go(Long x) { System.out.println("Long"); }

  public static void main(String [] args) {
    byte b = 5;
    go(b);           // must widen then box - illegal
  }
}
```

This is just too much for the compiler:

```
WidenAndBox.java:6: go(java.lang.Long) in WidenAndBox cannot be
applied to (byte)
```

Strangely enough, it IS possible for the compiler to perform a boxing operation followed by a widening operation in order to match an invocation to a method. This one might blow your mind:

```
class BoxAndWiden {
  static void go(Object o) {
    Byte b2 = (Byte) o;       // ok - it's a Byte object
    System.out.println(b2);
  }

  public static void main(String [] args) {
    byte b = 5;
    go(b);       // can this byte turn into an Object ?
  }
}
```

This compiles (!), and produces the output:

```
5
```

Wow! Here's what happened under the covers when the compiler, then the JVM, got to the line that invokes the `go()` method:

1. The `byte b` was boxed to a Byte.
2. The Byte reference was widened to an Object (since Byte extends Object).
3. The `go()` method got an Object reference that actually refers to a Byte object.
4. The `go()` method cast the Object reference back to a Byte reference (remember, there was never an object of type Object in this scenario, only an object of type Byte!).
5. The `go()` method printed the Byte's value.

Why didn't the compiler try to use the box-then-widen logic when it tried to deal with the WidenAndBox class? Think about it…if it tried to box first, the byte would have been converted to a Byte. Now we're back to trying to widen a Byte to a Long, and of course, the IS-A test fails.

Overloading in Combination with Var-args

What happens when we attempt to combine var-args with either widening or boxing in a method-matching scenario? Let's take a look:

```
class Vararg {
  static void wide_vararg(long... x)
              { System.out.println("long..."); }
  static void box_vararg(Integer... x)
              { System.out.println("Integer..."); }
  public static void main(String [] args) {
    int i = 5;
    wide_vararg(5,5);    // needs to widen and use var-args
    box_vararg(5,5);     // needs to box and use var-args
  }
}
```

This compiles and produces:

```
long...
Integer...
```

As we can see, you can successfully combine var-args with either widening or boxing. Here's a review of the rules for overloading methods using widening, boxing, and var-args:

- Primitive widening uses the "smallest" method argument possible.
- Used individually, boxing and var-args are compatible with overloading.
- You CANNOT widen from one wrapper type to another. (IS-A fails.)
- You CANNOT widen and then box. (An `int` can't become a Long.)
- You can box and then widen. (An `int` can become an Object, via Integer.)
- You can combine var-args with either widening or boxing.

There are more tricky aspects to overloading, but other than a few rules concerning generics (which we'll cover in Chapter 7), this is all you'll need to know for the exam. Phew!

CERTIFICATION OBJECTIVE

Garbage Collection (Exam Objective 7.4)

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.

Overview of Memory Management and Garbage Collection

This is the section you've been waiting for! It's finally time to dig into the wonderful world of memory management and garbage collection.

Memory management is a crucial element in many types of applications. Consider a program that reads in large amounts of data, say from somewhere else on a network, and then writes that data into a database on a hard drive. A typical design would be to read the data into some sort of collection in memory, perform some operations on the data, and then write the data into the database. After the data is written into the database, the collection that stored the data temporarily must be emptied of old data or deleted and re-created before processing the next

batch. This operation might be performed thousands of times, and in languages like C or C++ that do not offer automatic garbage collection, a small flaw in the logic that manually empties or deletes the collection data structures can allow small amounts of memory to be improperly reclaimed or lost. Forever. These small losses are called memory leaks, and over many thousands of iterations they can make enough memory inaccessible that programs will eventually crash. Creating code that performs manual memory management cleanly and thoroughly is a nontrivial and complex task, and while estimates vary, it is arguable that manual memory management can double the development effort for a complex program.

Java's garbage collector provides an automatic solution to memory management. In most cases it frees you from having to add any memory management logic to your application. The downside to automatic garbage collection is that you can't completely control when it runs and when it doesn't.

Overview of Java's Garbage Collector

Let's look at what we mean when we talk about garbage collection in the land of Java. From the 30,000 ft. level, garbage collection is the phrase used to describe automatic memory management in Java. Whenever a software program executes (in Java, C, C++, Lisp, Ruby, and so on), it uses memory in several different ways. We're not going to get into Computer Science 101 here, but it's typical for memory to be used to create a stack, a heap, in Java's case constant pools, and method areas. The heap is that part of memory where Java objects live, and it's the one and only part of memory that is in any way involved in the garbage collection process.

> A heap is a heap is a heap. For the exam it's important to know that you can call it the heap, you can call it the garbage collectible heap, you can call it Johnson, but there is one and only one heap.

So, all of garbage collection revolves around making sure that the heap has as much free space as possible. For the purpose of the exam, what this boils down to is deleting any objects that are no longer reachable by the Java program running. We'll talk more about what reachable means, but let's drill this point in. When the garbage collector runs, its purpose is to find and delete objects that cannot be reached. If you think of a Java program as being in a constant cycle of creating the objects it needs (which occupy space on the heap), and then discarding them when they're no longer needed, creating new objects, discarding them, and so on, the missing piece of the puzzle is the garbage collector. When it runs, it looks for those

discarded objects and deletes them from memory so that the cycle of using memory and releasing it can continue. Ah, the great circle of life.

When Does the Garbage Collector Run?

The garbage collector is under the control of the JVM. The JVM decides when to run the garbage collector. From within your Java program you can ask the JVM to run the garbage collector, but there are no guarantees, under any circumstances, that the JVM will comply. Left to its own devices, the JVM will typically run the garbage collector when it senses that memory is running low. Experience indicates that when your Java program makes a request for garbage collection, the JVM will usually grant your request in short order, but there are no guarantees. Just when you think you can count on it, the JVM will decide to ignore your request.

How Does the Garbage Collector Work?

You just can't be sure. You might hear that the garbage collector uses a mark and sweep algorithm, and for any given Java implementation that might be true, but the Java specification doesn't guarantee any particular implementation. You might hear that the garbage collector uses reference counting; once again maybe yes maybe no. The important concept to understand for the exam is when does an object become eligible for garbage collection? To answer this question fully, we have to jump ahead a little bit and talk about threads. (See Chapter 9 for the real scoop on threads.) In a nutshell, every Java program has from one to many threads. Each thread has its own little execution stack. Normally, you (the programmer) cause at least one thread to run in a Java program, the one with the `main()` method at the bottom of the stack. However, as you'll learn in excruciating detail in Chapter 9, there are many really cool reasons to launch additional threads from your initial thread. In addition to having its own little execution stack, each thread has its own lifecycle. For now, all we need to know is that threads can be alive or dead. With this background information, we can now say with stunning clarity and resolve that *an object is eligible for garbage collection when no live thread can access it.* (Note: Due to the vagaries of the String constant pool, the exam focuses its garbage collection questions on non-String objects, and so our garbage collection discussions apply to only non-String objects too.)

Based on that definition, the garbage collector does some magical, unknown operations, and when it discovers an object that can't be reached by any live thread, it will consider that object as eligible for deletion, and it might even delete it at some point. (You guessed it; it also might not ever delete it.) When we talk about reaching an object, we're really talking about having a reachable reference variable

that refers to the object in question. If our Java program has a reference variable that refers to an object, and that reference variable is available to a live thread, then that object is considered reachable. We'll talk more about how objects can become unreachable in the following section.

Can a Java application run out of memory? Yes. The garbage collection system attempts to remove objects from memory when they are not used. However, if you maintain too many live objects (objects referenced from other live objects), the system can run out of memory. Garbage collection cannot ensure that there is enough memory, only that the memory that is available will be managed as efficiently as possible.

Writing Code That Explicitly Makes Objects Eligible for Collection

In the preceding section, we learned the theories behind Java garbage collection. In this section, we show how to make objects eligible for garbage collection using actual code. We also discuss how to attempt to force garbage collection if it is necessary, and how you can perform additional cleanup on objects before they are removed from memory.

Nulling a Reference

As we discussed earlier, an object becomes eligible for garbage collection when there are no more reachable references to it. Obviously, if there are no reachable references, it doesn't matter what happens to the object. For our purposes it is just floating in space, unused, inaccessible, and no longer needed.

The first way to remove a reference to an object is to set the reference variable that refers to the object to `null`. Examine the following code:

```
1. public class GarbageTruck {
2.   public static void main(String [] args) {
3.     StringBuffer sb = new StringBuffer("hello");
4.     System.out.println(sb);
5.     // The StringBuffer object is not eligible for collection
6.     sb = null;
7.     // Now the StringBuffer object is eligible for collection
8.   }
9. }
```

The StringBuffer object with the value `hello` is assigned to the reference variable `sb` in the third line. To make the object eligible (for GC), we set the reference variable `sb` to `null`, which removes the single reference that existed to the

StringBuffer object. Once line 6 has run, our happy little `hello` StringBuffer object is doomed, eligible for garbage collection.

Reassigning a Reference Variable

We can also decouple a reference variable from an object by setting the reference variable to refer to another object. Examine the following code:

```
class GarbageTruck {
   public static void main(String [] args) {
      StringBuffer s1 = new StringBuffer("hello");
      StringBuffer s2 = new StringBuffer("goodbye");
      System.out.println(s1);
      // At this point the StringBuffer "hello" is not eligible
      s1 = s2; // Redirects s1 to refer to the "goodbye" object
      // Now the StringBuffer "hello" is eligible for collection
   }
}
```

Objects that are created in a method also need to be considered. When a method is invoked, any local variables created exist only for the duration of the method. Once the method has returned, the objects created in the method are eligible for garbage collection. There is an obvious exception, however. If an object is returned from the method, its reference might be assigned to a reference variable in the method that called it; hence, it will not be eligible for collection. Examine the following code:

```
import java.util.Date;
public class GarbageFactory {
   public static void main(String [] args) {
      Date d = getDate();
      doComplicatedStuff();
      System.out.println("d = " + d);
   }

   public static Date getDate() {
      Date d2 = new Date();
      StringBuffer now = new StringBuffer(d2.toString());
      System.out.println(now);
      return d2;
   }
}
```

In the preceding example, we created a method called `getDate()` that returns a Date object. This method creates two objects: a Date and a StringBuffer containing the date information. Since the method returns the Date object, it will not be eligible for collection even after the method has completed. The StringBuffer object, though, will be eligible, even though we didn't explicitly set the `now` variable to `null`.

Isolating a Reference

There is another way in which objects can become eligible for garbage collection, even if they still have valid references! We call this scenario "islands of isolation."

A simple example is a class that has an instance variable that is a reference variable to another instance of the same class. Now imagine that two such instances exist and that they refer to each other. If all other references to these two objects are removed, then even though each object still has a valid reference, there will be no way for any live thread to access either object. When the garbage collector runs, it can *usually* discover any such islands of objects and remove them. As you can imagine, such islands can become quite large, theoretically containing hundreds of objects. Examine the following code:

```
public class Island {
  Island i;
  public static void main(String [] args) {

    Island i2 = new Island();
    Island i3 = new Island();
    Island i4 = new Island();

    i2.i = i3;   // i2 refers to i3
    i3.i = i4;   // i3 refers to i4
    i4.i = i2;   // i4 refers to i2

    i2 = null;
    i3 = null;
    i4 = null;

    // do complicated, memory intensive stuff
  }
}
```

When the code reaches `// do complicated`, the three Island objects (previously known as `i2`, `i3`, and `i4`) have instance variables so that they refer to

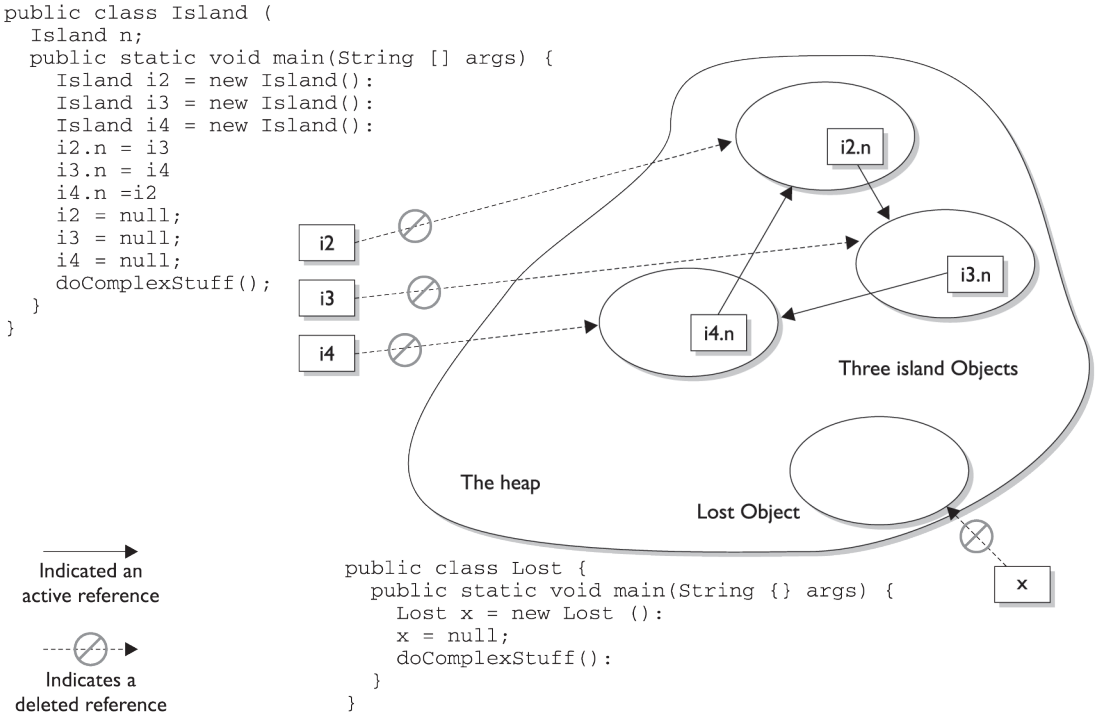
each other, but their links to the outside world (`i2`, `i3`, and `i4`) have been nulled. These three objects are eligible for garbage collection.

This covers everything you will need to know about making objects eligible for garbage collection. Study Figure 3-7 to reinforce the concepts of objects without references and islands of isolation.

Forcing Garbage Collection

The first thing that should be mentioned here is that, contrary to this section's title, garbage collection cannot be forced. However, Java provides some methods that allow you to request that the JVM perform garbage collection. For example, if you are about to perform some time-sensitive operations, you probably want to minimize the chances of a delay caused by garbage collection. But you must remember that the methods that Java provides are requests, and not demands; the virtual machine will do its best to do what you ask, but there is no guarantee that it will comply.

FIGURE 3-7 "Island" objects eligible for garbage collection



In reality, it is possible only to suggest to the JVM that it perform garbage collection. However, there are no guarantees the JVM will actually remove all of the unused objects from memory (even if garbage collection is run). It is essential that you understand this concept for the exam.

The garbage collection routines that Java provides are members of the Runtime class. The Runtime class is a special class that has a single object (a Singleton) for each main program. The Runtime object provides a mechanism for communicating directly with the virtual machine. To get the Runtime instance, you can use the method Runtime.getRuntime(), which returns the Singleton. Once you have the Singleton you can invoke the garbage collector using the `gc()` method. Alternatively, you can call the same method on the System class, which has static methods that can do the work of obtaining the Singleton for you. The simplest way to ask for garbage collection (remember—just a request) is

```
System.gc();
```

Theoretically, after calling `System.gc()`, you will have as much free memory as possible. We say theoretically because this routine does not always work that way. First, your JVM may not have implemented this routine; the language specification allows this routine to do nothing at all. Second, another thread (again, see the Chapter 9) might grab lots of memory right after you run the garbage collector.

This is not to say that `System.gc()` is a useless method—it's much better than nothing. You just can't rely on `System.gc()` to free up enough memory so that you don't have to worry about running out of memory. The Certification Exam is interested in guaranteed behavior, not probable behavior.

Now that we are somewhat familiar with how this works, let's do a little experiment to see if we can see the effects of garbage collection. The following program lets us know how much total memory the JVM has available to it and how much free memory it has. It then creates 10,000 Date objects. After this, it tells us how much memory is left and then calls the garbage collector (which, if it decides to run, should halt the program until all unused objects are removed). The final free memory result should indicate whether it has run. Let's look at the program:

```
1.  import java.util.Date;
2.  public class CheckGC {
3.     public static void main(String [] args) {
4.        Runtime rt = Runtime.getRuntime();
5.        System.out.println("Total JVM memory: "
                              + rt.totalMemory());
```

```
 6.         System.out.println("Before Memory = "
                               + rt.freeMemory());
 7.         Date d = null;
 8.         for(int i = 0;i<10000;i++) {
 9.            d = new Date();
10.            d = null;
11.         }
12.         System.out.println("After Memory = "
                               + rt.freeMemory());
13.         rt.gc();   // an alternate to System.gc()
14.         System.out.println("After GC Memory = "
                               + rt.freeMemory());
15.      }
16.   }
```

Now, let's run the program and check the results:

```
Total JVM memory: 1048568
Before Memory = 703008
After Memory = 458048
After GC Memory = 818272
```

As we can see, the JVM actually did decide to garbage collect (i.e., delete) the eligible objects. In the preceding example, we suggested to the JVM to perform garbage collection with 458,048 bytes of memory remaining, and it honored our request. This program has only one user thread running, so there was nothing else going on when we called `rt.gc()`. Keep in mind that the behavior when `gc()` is called may be different for different JVMs, so there is no guarantee that the unused objects will be removed from memory. About the only thing you can guarantee is that if you are running very low on memory, the garbage collector will run before it throws an OutOfMemoryException.

EXERCISE 3-2

Try changing the CheckGC program by putting lines 13 and 14 inside a loop. You might see that not all memory is released on any given run of the GC.

Cleaning Up Before Garbage Collection—the finalize() Method

Java provides you a mechanism to run some code just before your object is deleted by the garbage collector. This code is located in a method named `finalize()` that all classes inherit from class Object. On the surface this sounds like a great idea; maybe your object opened up some resources, and you'd like to close them before your object is deleted. The problem is that, as you may have gathered by now, you can't count on the garbage collector to ever delete an object. So, any code that you put into your class's overridden `finalize()` method is not guaranteed to run. The `finalize()` method for any given object might run, but you can't count on it, so don't put any essential code into your `finalize()` method. In fact, we recommend that in general you don't override `finalize()` at all.

Tricky Little finalize() Gotcha's

There are a couple of concepts concerning `finalize()` that you need to remember.

- For any given object, `finalize()` will be called only once (at most) by the garbage collector.
- Calling `finalize()` can actually result in saving an object from deletion.

Let's look into these statements a little further. First of all, remember that any code that you can put into a normal method you can put into `finalize()`. For example, in the `finalize()` method you could write code that passes a reference to the object in question back to another object, effectively *uneligiblizing* the object for garbage collection. If at some point later on this same object becomes eligible for garbage collection again, the garbage collector can still process this object and delete it. The garbage collector, however, will remember that, for this object, `finalize()` already ran, and it will not run `finalize()` again.

CERTIFICATION SUMMARY

This was a monster chapter! Don't worry if you find that you have to review some of these topics as you get into later chapters. This chapter has a lot of foundation stuff that will come into play later.

We started the chapter by reviewing the stack and the heap; remember local variables live on the stack, and instance variables live with their objects on the heap.

We reviewed legal literals for primitives and Strings, then we discussed the basics of assigning values to primitives and reference variables, and the rules for casting primitives.

Next we discussed the concept of scope, or "How long will this variable live?" Remember the four basic scopes, in order of lessening lifespan: static, instance, local, block.

We covered the implications of using uninitialized variables, and the importance of the fact that local variables MUST be assigned a value explicitly. We talked about some of the tricky aspects of assigning one reference variable to another, and some of the finer points of passing variables into methods, including a discussion of "shadowing."

The next topic was creating arrays, where we talked about declaring, constructing, and initializing one-, and multi-dimensional arrays. We talked about anonymous arrays, and arrays of references.

Next we reviewed static and instance initialization blocks, what they look like, and when they are called.

Phew!

We continued the chapter with a discussion of the wrapper classes; used to create immutable objects that hold a primitive, and also used to provide conversion capabilities for primitives: remember valueOf(), xxxValue(), and parseXxx().

Closely related to wrappers, we talked about a big new feature in Java 5, autoboxing. Boxing is a way to automate the use of wrappers, and we covered some of its trickier aspects such as how wrappers work with == and the equals() method.

Having added boxing to our toolbox, it was time to take a closer look at method overloading and how boxing and var-args, in conjunction with widening conversions, make overloading more complicated.

Finally, we dove into garbage collection, Java's automatic memory management feature. We learned that the heap is where objects live and where all the cool garbage collection activity takes place. We learned that in the end, the JVM will perform garbage collection whenever it wants to. You (the programmer) can request a garbage collection run, but you can't force it. We talked about garbage collection only applying to objects that are eligible, and that eligible means "inaccessible from any live thread." Finally, we discussed the rarely useful finalize() method, and what you'll have to know about it for the exam. All in all, one fascinating chapter.

TWO-MINUTE DRILL

Stack and Heap

- ❑ Local variables (method variables) live on the stack.
- ❑ Objects and their instance variables live on the heap.

Literals and Primitive Casting (Objective 1.3)

- ❑ Integer literals can be decimal, octal (e.g. `013`), or hexadecimal (e.g. `0x3d`).
- ❑ Literals for `long`s end in `L` or `l`.
- ❑ Float literals end in `F` or `f`, `double` literals end in a digit or `D` or `d`.
- ❑ The `boolean` literals are `true` and `false`.
- ❑ Literals for `char`s are a single character inside single quotes: `'d'`.

Scope (Objectives 1.3 and 7.6)

- ❑ Scope refers to the lifetime of a variable.
- ❑ There are four basic scopes:
  - ❑ Static variables live basically as long as their class lives.
  - ❑ Instance variables live as long as their object lives.
  - ❑ Local variables live as long as their method is on the stack; however, if their method invokes another method, they are temporarily unavailable.
  - ❑ Block variables (e.g.., in a `for` or an `if`) live until the block completes.

Basic Assignments (Objectives 1.3 and 7.6)

- ❑ Literal integers are implicitly ints.
- ❑ Integer expressions always result in an `int`-sized result, never smaller.
- ❑ Floating-point numbers are implicitly doubles (64 bits).
- ❑ Narrowing a primitive truncates the *high order* bits.
- ❑ Compound assignments (e.g. `+=`), perform an automatic cast.
- ❑ A reference variable holds the bits that are used to refer to an object.
- ❑ Reference variables can refer to subclasses of the declared type but not to superclasses.

- When creating a new object, e.g., `Button b = new Button();`, three things happen:
  - Make a reference variable named `b`, of type Button
  - Create a new Button object
  - Assign the Button object to the reference variable `b`

Using a Variable or Array Element That Is Uninitialized and Unassigned (Objectives 1.3 and 7.6)

- 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, elements get default values.
- 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.

Passing Variables into Methods (Objective 7.3)

- Methods can take primitives and/or object references as arguments.
- Method arguments are always copies.
- Method arguments are never actual objects (they can be references to objects).
- A primitive argument is an unattached copy of the original primitive.
- A reference argument is another copy of a reference to the original object.
- Shadowing occurs when two variables with different scopes share the same name. This leads to hard-to-find bugs, and hard-to-answer exam questions.

Array Declaration, Construction, and Initialization (Obj. 1.3)

- Arrays can hold primitives or objects, but the array itself is always an object.
- When you declare an array, the brackets can be left or right of the 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.
- ❑ An ArrayIndexOutOfBoundsException occurs if you use a bad index value.
- ❑ Arrays have a `length` variable whose value is the number of array elements.
- ❑ 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 array's declared type;. e.g., a `byte` variable can go 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).

Initialization Blocks (Objectives 1.3 and 7.6)

- ❑ Static initialization blocks run once, when the class is first loaded.
- ❑ Instance initialization blocks run every time a new instance is created. They run after all super-constructors and before the constructor's code has run.
- ❑ If multiple init blocks exist in a class, they follow the rules stated above, AND they run in the order in which they appear in the source file.

Using Wrappers (Objective 3.1)

- ❑ The wrapper classes correlate to the primitive types.
- ❑ Wrappers have two main functions:
  - ❑ To wrap primitives so that they can be handled like objects
  - ❑ To provide utility methods for primitives (usually conversions)
- ❑ The three most important method families are
  - ❑ `xxxValue()` Takes no arguments, returns a primitive
  - ❑ `parseXxx()` Takes a String, returns a primitive, throws NFE
  - ❑ `valueOf()` Takes a String, returns a wrapped object, throws NFE

- ❑ Wrapper constructors can take a String or a primitive, except for Character, which can only take a `char`.
- ❑ Radix refers to bases (typically) other than 10; octal is radix = 8, hex = 16.

Boxing (Objective 3.1)

- ❑ As of Java 5, boxing allows you to convert primitives to wrappers or to convert wrappers to primitives automatically.
- ❑ Using == with wrappers is tricky; wrappers with the same small values (typically lower than 127), will be ==, larger values will not be ==.

Advanced Overloading (Objectives 1.5 and 5.4)

- ❑ Primitive widening uses the "smallest" method argument possible.
- ❑ Used individually, boxing and var-args are compatible with overloading.
- ❑ You CANNOT widen from one wrapper type to another. (IS-A fails.)
- ❑ You CANNOT widen and then box. (An `int` can't become a Long.)
- ❑ You can box and then widen. (An `int` can become an Object, via an Integer.)
- ❑ You can combine var-args with either widening or boxing.

Garbage Collection (Objective 7.4)

- ❑ In Java, garbage collection (GC) provides automated memory management.
- ❑ The purpose of GC is to delete objects that can't be reached.
- ❑ Only the JVM decides when to run the GC, you can only suggest it.
- ❑ You can't know the GC algorithm for sure.
- ❑ Objects must be considered eligible before they can be garbage collected.
- ❑ An object is eligible when no live thread can reach it.
- ❑ To reach an object, you must have a live, reachable reference to that object.
- ❑ Java applications can run out of memory.
- ❑ Islands of objects can be GCed, even though they refer to each other.
- ❑ Request garbage collection with `System.gc();` (recommended).
- ❑ Class Object has a `finalize()` method.
- ❑ The `finalize()` method is guaranteed to run once and only once before the garbage collector deletes an object.
- ❑ The garbage collector makes no guarantees, `finalize()` may never run.
- ❑ You can uneligibilize an object for GC from within `finalize()`.

SELF TEST

1. Given:

```
class Scoop {
  static int thrower() throws Exception { return 42; }
  public static void main(String [] args) {
    try {
      int x = thrower();
    } catch (Exception e) {
      x++;
    } finally {
      System.out.println("x = " + ++x);
} } }
```

What is the result?

A. `x = 42`

B. `x = 43`

C. `x = 44`

D. Compilation fails.

E. The code runs with no output.

2. Given:

```
class CardBoard {
  Short story = 5;
  CardBoard go(CardBoard cb) {
    cb = null;
    return cb;
  }
  public static void main(String[] args) {
    CardBoard c1 = new CardBoard();
    CardBoard c2 = new CardBoard();
    CardBoard c3 = c1.go(c2);
    c1 = null;
    // do Stuff
} }
```

When `// doStuff` is reached, how many objects are eligible for GC?

A. 0

B. 1

C. 2

D. Compilation fails.

E. It is not possible to know.

F. An exception is thrown at runtime.

3. Given:

```
class Alien {
  String invade(short ships) { return "a few"; }
  String invade(short... ships) { return "many"; }
}
class Defender {
  public static void main(String [] args) {
    System.out.println(new Alien().invade(7));
  }
}
```

What is the result?

A. `many`
B. `a few`
C. Compilation fails.
D. The output is not predictable.
E. An exception is thrown at runtime.

4. Given:

```
1. class Dims {
2.   public static void main(String[] args) {
3.     int[][] a = {{1,2,}, {3,4}};
4.     int[] b = (int[]) a[1];
5.     Object o1 = a;
6.     int[][] a2 = (int[][]) o1;
7.     int[] b2 = (int[]) o1;
8.     System.out.println(b[1]);
9.   }
10. }
```

What is the result?

A. 2
B. 4
C. An exception is thrown at runtime.
D. Compilation fails due to an error on line 4.
E. Compilation fails due to an error on line 5.
F. Compilation fails due to an error on line 6.
G. Compilation fails due to an error on line 7.

5. Given:

```
class Eggs {
  int doX(Long x, Long y) { return 1; }
  int doX(long... x) { return 2; }
  int doX(Integer x, Integer y) { return 3; }
  int doX(Number n, Number m) { return 4; }
  public static void main(String[] args) {
    new Eggs().go();
  }
  void go() {
    short s = 7;
    System.out.print(doX(s,s) + " ");
    System.out.println(doX(7,7));
} }
```

What is the result?

A. `1 1`
B. `2 1`
C. `3 1`
D. `4 1`
E. `2 3`
F. `3 3`
G. `4 3`

6. Given:

```
class Mixer {
  Mixer() { }
  Mixer(Mixer m) { m1 = m; }
  Mixer m1;
  public static void main(String[] args) {
    Mixer m2 = new Mixer();
    Mixer m3 = new Mixer(m2);  m3.go();
    Mixer m4 = m3.m1;          m4.go();
    Mixer m5 = m2.m1;          m5.go();
  }
  void go() { System.out.print("hi "); }
}
```

What is the result?

A. `hi`
B. `hi hi`
C. `hi hi hi`
D. Compilation fails
E. `hi`, followed by an exception
F. `hi hi`, followed by an exception

7. Given:

```
1. class Zippy {
2.   String[] x;
3.   int[] a [] = {{1,2}, {1}};
4.   Object c = new long[4];
5.   Object[] d = x;
6. }
```

What is the result?

A. Compilation succeeds.
B. Compilation fails due only to an error on line 3.
C. Compilation fails due only to an error on line 4.
D. Compilation fails due only to an error on line 5.
E. Compilation fails due to errors on lines 3 and 5.
F. Compilation fails due to errors on lines 3, 4, and 5.

8. Given:

```
class Fizz {
  int x = 5;
  public static void main(String[] args) {
    final Fizz f1 = new Fizz();
    Fizz f2 = new Fizz();
    Fizz f3 = FizzSwitch(f1,f2);
    System.out.println((f1 == f3) + " " + (f1.x == f3.x));
  }
  static Fizz FizzSwitch(Fizz x, Fizz y) {
    final Fizz z = x;
    z.x = 6;
    return z;
} }
```

What is the result?

A. `true true`
B. `false true`
C. `true false`
D. `false false`
E. Compilation fails.
F. An exception is thrown at runtime.

9. Given:

```
class Knowing {
  static final long tooth = 343L;
  static long doIt(long tooth) {
    System.out.print(++tooth + " ");
```

```
    return ++tooth;
  }
  public static void main(String[] args) {
    System.out.print(tooth + " ");
    final long tooth = 340L;
    new Knowing().doIt(tooth);
    System.out.println(tooth);
  }
}
```

What is the result?

A. `343 340 340`

B. `343 340 342`

C. `343 341 342`

D. `343 341 340`

E. `343 341 343`

F. Compilation fails.

G. An exception is thrown at runtime.

10. Which is true? (Choose all that apply.)

A. The invocation of an object's `finalize()` method is always the last thing that happens before an object is garbage collected (GCed).

B. When a stack variable goes out of scope it is eligible for GC.

C. Some reference variables live on the stack, and some live on the heap.

D. Only objects that have no reference variables referring to them can be eligible for GC.

E. It's possible to request the GC via methods in either `java.lang.Runtime` or `java.lang.System` classes.

11. Given:

```
1. class Convert {
2.   public static void main(String[] args) {
3.     Long xL = new Long(456L);
4.     long x1 = Long.valueOf("123");
5.     Long x2 = Long.valueOf("123");
6.     long x3 = xL.longValue();
7.     Long x4 = xL.longValue();
8.     Long x5 = Long.parseLong("456");
9.     long x6 = Long.parseLong("123");
10.  }
11. }
```

Which will compile using Java 5, but will NOT compile using Java 1.4? (Choose all that apply.)

A. Line 4
B. Line 5
C. Line 6
D. Line 7
E. Line 8
F. Line 9

12. Given:

```
1. class Eco {
2.   public static void main(String[] args) {
3.     Eco e1 = new Eco();
4.     Eco e2 = new Eco();
5.     Eco e3 = new Eco();
6.     e3.e = e2;
7.     e1.e = e3;
8.     e2 = null;
9.     e3 = null;
10.    e2.e = e1;
11.    e1 = null;
12.  }
13.  Eco e;
14. }
```

At what point is only a single object eligible for GC?

A. After line 8 runs.
B. After line 9 runs.
C. After line 10 runs.
D. After line 11 runs.
E. Compilation fails.
F. Never in this program.
G. An exception is thrown at runtime.

13. Given:

```
1. class Bigger {
2.   public static void main(String[] args) {
3.   // insert code here
4.   }
5. }
6. class Better {
7.   enum Faster {Higher, Longer};
8. }
```

Which, inserted independently at line 3, will compile? (Choose all that apply.)

A. `Faster f = Faster.Higher;`

B. `Faster f = Better.Faster.Higher;`

C. `Better.Faster f = Better.Faster.Higher;`

D. `Bigger.Faster f = Bigger.Faster.Higher;`

E. `Better.Faster f2;  f2 = Better.Faster.Longer;`

F. `Better b;  b.Faster = f3;  f3 = Better.Faster.Longer;`

14. Given:

```
class Bird {
  { System.out.print("b1 "); }
  public Bird() { System.out.print("b2 "); }
}
class Raptor extends Bird {
  static { System.out.print("r1 "); }
  public Raptor() { System.out.print("r2 "); }
  { System.out.print("r3 "); }
  static { System.out.print("r4 "); }
}
class Hawk extends Raptor {
  public static void main(String[] args) {
    System.out.print("pre ");
    new Hawk();
    System.out.println("hawk ");
  }
}
```

What is the result?

A. `pre b1 b2 r3 r2 hawk`

B. `pre b2 b1 r2 r3 hawk`

C. `pre b2 b1 r2 r3 hawk r1 r4`

D. `r1 r4 pre b1 b2 r3 r2 hawk`

E. `r1 r4 pre b2 b1 r2 r3 hawk`

F. `pre r1 r4 b1 b2 r3 r2 hawk`

G. `pre r1 r4 b2 b1 r2 r3 hawk`

H. The order of output cannot be predicted.

I. Compilation fails.

SELF TEST ANSWERS

1. Given:

```
class Scoop {
  static int thrower() throws Exception { return 42; }
  public static void main(String [] args) {
    try {
      int x = thrower();
    } catch (Exception e) {
      x++;
    } finally {
      System.out.println("x = " + ++x);
} } }
```

What is the result?

A. `x = 42`

B. `x = 43`

C. `x = 44`

D. Compilation fails.

E. The code runs with no output.

Answer:

☑ **D** is correct, the variable `x` is only in scope within the `try` code block, it's not in scope in the `catch` or `finally` blocks. (For the exam, get used to those horrible closing `} } } .`)

☒ **A, B, C,** and **E** is are incorrect based on the above. (Objective 1.3)

2. Given:

```
class CardBoard {
  Short story = 5;
  CardBoard go(CardBoard cb) {
    cb = null;
    return cb;
  }
  public static void main(String[] args) {
    CardBoard c1 = new CardBoard();
    CardBoard c2 = new CardBoard();
    CardBoard c3 = c1.go(c2);
    c1 = null;
    // do Stuff
} }
```

When `// doStuff` is reached, how many objects are eligible for GC?

A. 0

B. 1

C. 2
D. Compilation fails.
E. It is not possible to know.
F. An exception is thrown at runtime.

Answer:

☑ **C** is correct. Only one `CardBoard object (c1)` is eligible, but it has an associated Short wrapper object that is also eligible.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 7.4)

3. Given:

```
class Alien {
  String invade(short ships) { return "a few"; }
  String invade(short... ships) { return "many"; }
}
class Defender {
  public static void main(String [] args) {
    System.out.println(new Alien().invade(7));
} }
```

What is the result?

A. `many`
B. `a few`
C. Compilation fails.
D. The output is not predictable.
E. An exception is thrown at runtime.

Answer:

☑ **C** is correct, compilation fails. The var-args declaration is fine, but `invade` takes a `short`, so the argument 7 needs to be cast to a `short`. With the cast, the answer is **B**, '`a few`'.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objective 1.3)

4. Given:

```
1. class Dims {
2.   public static void main(String[] args) {
3.     int[][] a = {{1,2,}, {3,4}};
4.     int[] b = (int[]) a[1];
5.     Object o1 = a;
6.     int[][] a2 = (int[][]) o1;
7.     int[] b2 = (int[]) o1;
8.     System.out.println(b[1]);
9. } }
```

What is the result?

A. `2`
B. `4`
C. An exception is thrown at runtime
D. Compilation fails due to an error on line 4.
E. Compilation fails due to an error on line 5.
F. Compilation fails due to an error on line 6.
G. Compilation fails due to an error on line 7.

Answer:

☑ **C** is correct. A `ClassCastException` is thrown at line 7 because `o1` refers to an `int[][]` not an `int[]`. If line 7 was removed, the output would be 4.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (Objective 1.3)

5. Given:

```
class Eggs {
  int doX(Long x, Long y) { return 1; }
  int doX(long... x) { return 2; }
  int doX(Integer x, Integer y) { return 3; }
  int doX(Number n, Number m) { return 4; }
  public static void main(String[] args) {
    new Eggs().go();
  }
  void go() {
    short s = 7;
    System.out.print(doX(s,s) + " ");
    System.out.println(doX(7,7));
} }
```

What is the result?

A. `1 1`
B. `2 1`
C. `3 1`
D. `4 1`
E. `2 3`
F. `3 3`
G. `4 3`

Answer:

☑ **G** is correct. Two rules apply to the first invocation of `doX()`. You can't widen and then box in one step, and var-args are always chosen last. Therefore you can't widen shorts to either ints or longs, and then box them to Integers or Longs. But you can box shorts to Shorts and

then widen them to Numbers, and this takes priority over using a var-args method. The second invocation uses a simple box from int to Integer.

☒ **A, B, C, D, E,** and **F** are incorrect based on the above. (Objective 3.1)

6. Given:

```
class Mixer {
  Mixer() { }
  Mixer(Mixer m) { m1 = m; }
  Mixer m1;
  public static void main(String[] args) {
    Mixer m2 = new Mixer();
    Mixer m3 = new Mixer(m2);  m3.go();
    Mixer m4 = m3.m1;          m4.go();
    Mixer m5 = m2.m1;          m5.go();
  }
  void go() { System.out.print("hi "); }
}
```

What is the result?

A. `hi`

B. `hi hi`

C. `hi hi hi`

D. Compilation fails

E. `hi`, followed by an exception

F. `hi hi`, followed by an exception

Answer:

☑ **F** is correct. The `m2` object's `m1` instance variable is never initialized, so when `m5` tries to use it a `NullPointerException` is thrown.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 7.3)

7. Given:

```
1. class Zippy {
2.   String[] x;
3.   int[] a [] = {{1,2}, {1}};
4.   Object c = new long[4];
5.   Object[] d = x;
6. }
```

What is the result?

A. Compilation succeeds.

B. Compilation fails due only to an error on line 3.

C. Compilation fails due only to an error on line 4.

D. Compilation fails due only to an error on line 5.

E. Compilation fails due to errors on lines 3 and 5.

F. Compilation fails due to errors on lines 3, 4, and 5.

Answer:

☑ **A** is correct, all of these array declarations are legal. Lines 4 and 5 demonstrate that arrays can be cast.

☒ **B, C, D, E,** and **F** are incorrect because this code compiles. (Objective 1.3)

8. Given:

```
class Fizz {
  int x = 5;
  public static void main(String[] args) {
    final Fizz f1 = new Fizz();
    Fizz f2 = new Fizz();
    Fizz f3 = FizzSwitch(f1,f2);
    System.out.println((f1 == f3) + " " + (f1.x == f3.x));
  }
  static Fizz FizzSwitch(Fizz x, Fizz y) {
    final Fizz z = x;
    z.x = 6;
    return z;
} }
```

What is the result?

A. `true true`

B. `false true`

C. `true false`

D. `false false`

E. Compilation fails.

F. An exception is thrown at runtime.

Answer:

☑ **A** is correct. The references `f1`, `z`, and `f3` all refer to the same instance of `Fizz`. The `final` modifier assures that a reference variable cannot be referred to a different object, but `final` doesn't keep the object's state from changing.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objective 7.3)

9. Given:

```
class Knowing {
  static final long tooth = 343L;
  static long doIt(long tooth) {
    System.out.print(++tooth + " ");
```

```
    return ++tooth;
  }
  public static void main(String[] args) {
    System.out.print(tooth + " ");
    final long tooth = 340L;
    new Knowing().doIt(tooth);
    System.out.println(tooth);
} }
```

What is the result?

A. `343 340 340`
B. `343 340 342`
C. `343 341 342`
D. `343 341 340`
E. `343 341 343`
F. Compilation fails.
G. An exception is thrown at runtime.

Answer:

☑ **D** is correct. There are three different `long` variables named tooth. Remember that you can apply the `final` modifier to local variables, but in this case the 2 versions of `tooth` marked `final` are not changed. The only `tooth` whose value changes is the one not marked `final`. This program demonstrates a bad practice known as shadowing.

☒ **A, B, C, E, F,** and **G** are incorrect based on the above. (Objective 7.3)

10. Which is true? (Choose all that apply.)

A. The invocation of an object's `finalize()` method is always the last thing that happens before an object is garbage collected (GCed).
B. When a stack variable goes out of scope it is eligible for GC.
C. Some reference variables live on the stack, and some live on the `heap`.
D. Only objects that have no reference variables referring to them can be eligible for GC.
E. It's possible to request the GC via methods in either `java.lang.Runtime` or `java.lang.System` classes.

Answer:

☑ **C** and **E** are correct. When an object has a reference variable, the reference variable lives inside the object, on the heap.

☒ **A** is incorrect, because if, the first time an object's `finalize()` method runs, the object is saved from the GC, then the second time that object is about to be GCed, `finalize()` will not run. **B** is incorrect—stack variables are not dealt with by the GC. **D** is incorrect because objects can live in "islands of isolation" and be GC eligible. (Objective 7.4)

11. Given:

```
1. class Convert {
2.   public static void main(String[] args) {
3.     Long xL = new Long(456L);
4.     long x1 = Long.valueOf("123");
5.     Long x2 = Long.valueOf("123");
6.     long x3 = xL.longValue();
7.     Long x4 = xL.longValue();
8.     Long x5 = Long.parseLong("456");
9.     long x6 = Long.parseLong("123");
10.  }
11. }
```

Which will compile using Java 5, but will NOT compile using Java 1.4? (Choose all that apply.)

A. Line 4.
B. Line 5.
C. Line 6.
D. Line 7.
E. Line 8.
F. Line 9.

Answer:

☑ **A, D,** and **E** are correct. Because of the methods' return types, these method calls required autoboxing to compile.

☒ **B, C,** and **F** are incorrect based on the above. (Objective 3.1)

12. Given:

```
1. class Eco {
2.   public static void main(String[] args) {
3.     Eco e1 = new Eco();
4.     Eco e2 = new Eco();
5.     Eco e3 = new Eco();
6.     e3.e = e2;
7.     e1.e = e3;
8.     e2 = null;
9.     e3 = null;
10.    e2.e = e1;
11.    e1 = null;
12.  }
13.  Eco e;
14. }
```

At what point is only a single object eligible for GC?

A. After line 8 runs.
B. After line 9 runs.
C. After line 10 runs.
D. After line 11 runs.
E. Compilation fails.
F. Never in this program.
G. An exception is thrown at runtime.

Answer:

☑ **G** is correct. An error at line 10 causes a `NullPointerException` to be thrown because `e2` was set to `null` in line 8. If line 10 was moved between lines 7 and 8, then **F** would be correct, because until the last reference is nulled none of the objects is eligible, and once the last reference is nulled, all three are eligible.

☒ **A, B, C, D, E,** and **F** are incorrect based on the above. (Objective 7.4)

13. Given:

```
1. class Bigger {
2.   public static void main(String[] args) {
3.   // insert code here
4.   }
5. }
6. class Better {
7.   enum Faster {Higher, Longer};
8. }
```

Which, inserted independently at line 3, will compile? (Choose all that apply.)

A. `Faster f = Faster.Higher;`
B. `Faster f = Better.Faster.Higher;`
C. `Better.Faster f = Better.Faster.Higher;`
D. `Bigger.Faster f = Bigger.Faster.Higher;`
E. `Better.Faster f2;  f2 = Better.Faster.Longer;`
F. `Better b;  b.Faster = f3;  f3 = Better.Faster.Longer;`

Answer:

☑ **C** and **E** are correct syntax for accessing an enum from another class.

☒ **A, B, D,** and **F** are incorrect syntax.
(Objective 1.3)

14. Given:

```
class Bird {
  { System.out.print("b1 "); }
  public Bird() { System.out.print("b2 "); }
}
class Raptor extends Bird {
  static { System.out.print("r1 "); }
  public Raptor() { System.out.print("r2 "); }
  { System.out.print("r3 "); }
  static { System.out.print("r4 "); }
}
class Hawk extends Raptor {
  public static void main(String[] args) {
    System.out.print("pre ");
    new Hawk();
    System.out.println("hawk ");
  }
}
```

What is the result?

A. `pre b1 b2 r3 r2 hawk`

B. `pre b2 b1 r2 r3 hawk`

C. `pre b2 b1 r2 r3 hawk r1 r4`

D. `r1 r4 pre b1 b2 r3 r2 hawk`

E. `r1 r4 pre b2 b1 r2 r3 hawk`

F. `pre r1 r4 b1 b2 r3 r2 hawk`

G. `pre r1 r4 b2 b1 r2 r3 hawk`

H. The order of output cannot be predicted.

I. Compilation fails.

Answer:

☑ **D** is correct. Static init blocks are executed at class loading time, instance init blocks run right after the call to `super()` in a constructor. When multiple init blocks of a single type occur in a class, they run in order, from the top down.

☒ **A, B, C, E, F, G, H,** and **I** are incorrect based on the above. Note: you'll probably never see this many choices on the real exam!
(Objective 1.3)