OCA Java SE 8 Programmer I Exam Guide (Exams 1Z0-808)

by Kathy Sierra and Bert Bates

Chapter 6: Strings, Arrays, ArrayLists, Dates, and Lambdas

CERTIFICATION OBJECTIVES

- Create and Manipulate Strings
- Manipulate Data Using the StringBuilder Class and Its Methods
- Create and Use Calendar Data
- Declare, Instantiate, Initialize, and Use a One-Dimensional Array
- Declare, Instantiate, Initialize, and Use a Multidimensional Array
- Declare and Use an ArrayList
- Use Wrapper Classes
- Use Encapsulation for Reference Variables
- Use Simple Lambda Expressions
- Two-Minute Drill
- Self Test

This chapter focuses on the exam objectives related to searching, formatting, and parsing strings; creating and using calendar-related objects; creating and using arrays and `ArrayLists`; and using simple lambda expressions. Many of these topics could fill an entire book. Fortunately, you won't have to become a total guru to do well on the exam. The exam team intended to include just the basic aspects of these technologies, and in this chapter, we cover *more* than you'll need to get through the related objectives on the exam.

CERTIFICATION OBJECTIVE: USING STRING AND STRINGBUILDER (OCA OBJECTIVES 9.2 AND 9.1)

9.2 Creating and manipulating Strings.

9.1 Manipulate data using the `StringBuilder` class and its methods.

Everything you needed to know about strings in the older OCJP exams you'll need to know for the OCA 8 exam. Closely related to the `String` class are the `StringBuilder` class and the almost identical `StringBuffer` class. (For the exam, the only thing you need to know about the `StringBuffer` class is that it has exactly the same methods as the `StringBuilder` class, but `StringBuilder` is faster because its methods aren't synchronized.) Both classes, `StringBuilder` and `StringBuffer`, give you `String`-like objects and ways to manipulate them, with the important difference being that these objects are mutable.

The String Class

This section covers the `String` class, and the key concept for you to understand is that once a `String` object is created, it can never be changed. So, then, what is happening when a `String` object seems to be changing? Let's find out.

Strings Are Immutable Objects

We'll start with a little background information about strings. You may not need this for the test, but a little context will help. Handling "strings" of characters is a fundamental aspect of most programming languages. In Java, each character in a string is a 16-bit Unicode character. Because Unicode characters are 16 bits (not the skimpy 7 or 8 bits that ASCII provides), a rich, international set of characters is easily represented in Unicode.

In Java, strings are objects. As with other objects, you can create an instance of a string with the `new` keyword, as follows:

```
String s = new String();
```

This line of code creates a new object of class `String` and assigns it to the reference variable `s`.

So far, `String` objects seem just like other objects. Now, let's give the string a value:

```
s = "abcdef";
```

(As you'll find out shortly, these two lines of code aren't quite what they seem, so stay tuned.)

It turns out the `String` class has about a zillion constructors, so you can use a more efficient shortcut:

```
String s = new String("abcdef");
```

And this is even more concise:

```
String s = "abcdef";
```

There are some subtle differences between these options that we'll discuss later, but what they have in common is that they all create a new `String` object, with a value of `"abcdef"`, and assign it to a reference variable `s`. Now let's say you want a second reference to the `String` object referred to by `s`:

```
String s2 = s; // refer s2 to the same String as s
```

So far so good. `String` objects seem to be behaving just like other objects, so what's all the fuss about? Immutability! (What the heck is immutability?) Once you have assigned a `String` a value, that value can never change—it's immutable, frozen solid, won't budge, *fini*, done. (We'll talk about why later; don't let us forget.) The good news is that although the `String` object is immutable, its reference variable is not, so to continue with our previous example, consider this:

```
s = s.concat(" more stuff"); // the concat() method 'appends'
                             // a literal to the end
```

Now, wait just a minute, didn't we just say that `String` objects were immutable? So what's all this "appending to the end of the string" talk? Excellent question: let's look at what really happened.

The Java Virtual Machine (JVM) took the value of string `s` (which was `"abcdef"`) and tacked `" more stuff"` onto the end, giving us the value `"abcdef more stuff"`. Since strings are immutable, the JVM couldn't stuff this new value into the old `String` referenced by `s`, so it created a new `String` object, gave it the value `"abcdef more stuff"`, and made `s` refer to it. At this point in our example, we have two `String` objects: the first one we created, with the value `"abcdef"`, and the second one with the value `"abcdef more stuff"`. Technically there are now three `String` objects, because the literal argument to `concat`, `" more stuff"`, is itself a new `String` object. But we have references only to `"abcdef"` (referenced by `s2`) and `"abcdef more stuff"` (referenced by `s`).

What if we didn't have the foresight or luck to create a second reference variable for the `"abcdef"` string before we called `s = s.concat (" more stuff");`? In that case, the original, unchanged string containing `"abcdef"` would still exist in memory, but it would be considered "lost." No code in our program has any way to reference it—it is lost to us. Note, however, that the original `"abcdef"` string didn't change (it can't, remember; it's immutable); only the reference variable `s` was changed so that it would refer to a different string.

Figure 6-1 shows what happens on the heap when you reassign a reference variable. Note that the dashed line indicates a deleted reference.

To review our first example:

```
String s = "abcdef"; // create a new String object, with
                     // value "abcdef", refer s to it
String s2 = s;       // create a 2nd reference variable
                     // referring to the same String

// create a new String object, with value "abcdef more stuff",
// refer s to it. (Change s's reference from the old String
// to the new String.) (Remember s2 is still referring to
// the original "abcdef" String.)

s = s.concat(" more stuff");
```

Let's look at another example:

```
String x = "Java";
x.concat(" Rules!");
System.out.println("x = " + x); // the output is "x = Java"
```

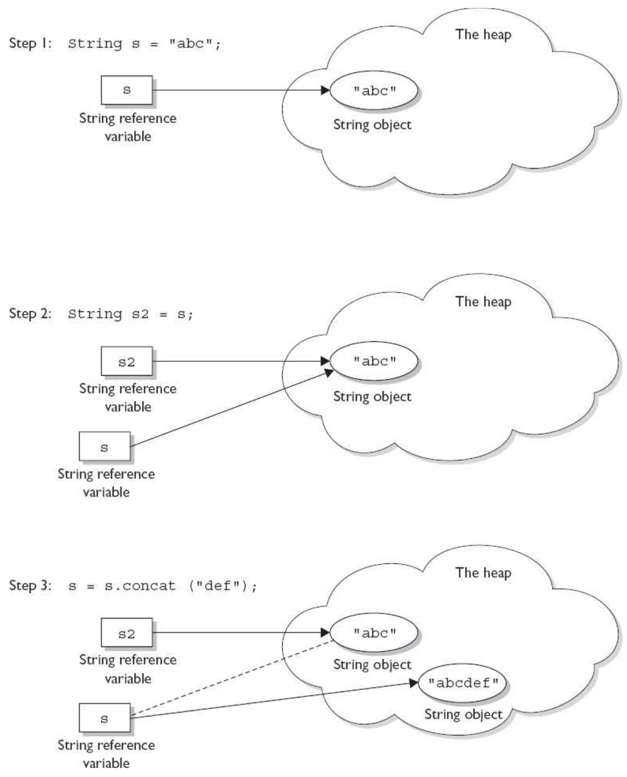

Step 1: String s = "abc";
The heap
s
String reference variable
"abc"
String object
Step 2: String s2 = s;
The heap
s2
String reference variable
"abc"
String object
s
String reference variable
Step 3: s = s.concat ("def");
The heap
s2
String reference variable
"abc"
String object
"abcdef"
String object
s
String reference variable

Figure 6-1: `String` objects and their reference variables

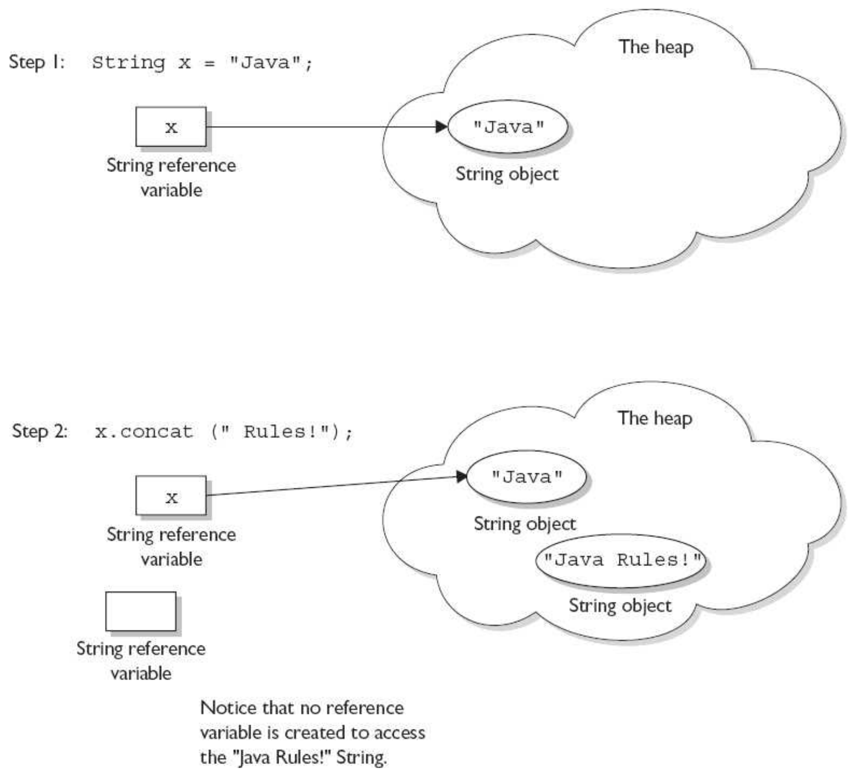

Step 1: String x = "Java";
The heap
x
String reference variable
"Java"
String object
Step 2: x.concat (" Rules!");
The heap
x
String reference variable
"Java"
String object
"Java Rules!"
String object
String reference variable
Notice that no reference variable is created to access the "Java Rules!" String.

Figure 6-2: A `String` object is abandoned upon creation.

The first line is straightforward: Create a new `String` object, give it the value `"Java",` and refer `x` to it. Next the JVM creates a second `String` object with the value `"Java Rules!"` but nothing refers to it. The second `String` object is instantly lost; you can't get to it. The reference variable `x` still refers to the original `String` with the value `"Java"`. Figure 6-2 shows creating a `String` without assigning a reference to it.

Let's expand this current example. We started with

```
String x = "Java";
x.concat(" Rules!");
```

```
System.out.println("x = " + x);    // the output is: x = Java
```

Now let's add

```
x.toUpperCase();
System.out.println("x = " + x);    // the output is still:
                                   // x = Java
```

(We actually did just create a new `String` object with the value `"JAVA"`, but it was lost, and `x` still refers to the original unchanged string `"Java"`.) How about adding this:

```
x.replace('a', 'X');
System.out.printsln("x = " + x);   // the output is still: x = Java
```

Can you determine what happened? The JVM created yet another new `String` object, with the value `"JXvX"`, (replacing the a's with `X`'s), but once again this new `String` was lost, leaving `x` to refer to the original unchanged and unchangeable `String` object, with the value `"Java"`. In all these cases, we called various string methods to create a new `String` by altering an existing `String`, but we never assigned the newly created `String` to a reference variable.

But we can put a small spin on the previous example:

```
String x = "Java";
x = x.concat(" Rules!");           // assign new string to x
System.out.println("x = " + x);    // output: x = Java Rules!
```

This time, when the JVM runs the second line, a new `String` object is created with the value `"Java Rules!"`, and `x` is set to reference it. But wait…there's more—now the original `String` object, `"Java"`, has been lost, and no one is referring to it. So in both examples, we created two `String` objects and only one reference variable, so one of the two `String` objects was left out in the cold. (See Figure 6-3 for a graphic depiction of this sad story.)

Let's take this example a little further:

```
String x = "Java";
x = x.concat(" Rules!");
System.out.println("x = " + x);         // output: x = Java Rules!

x.toLowerCase();                        // no assignment, create a
                                        // new, abandoned String

System.out.println("x = " + x);         // no assignment, the output
                                        // is still: x = Java Rules!

x = x.toLowerCase();                    // create a new String,
                                        // assigned to x
System.out.println("x = " + x);         // the assignment causes the
                                        // output: x = java rules!
```

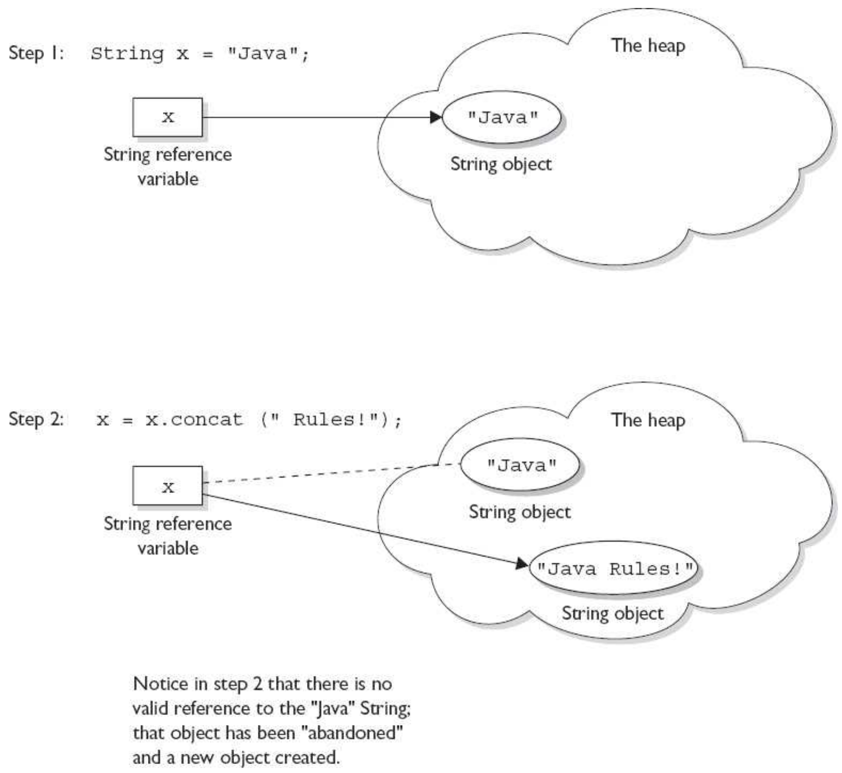

Step 1: String x = "Java";
x
String reference variable
The heap
"Java"
String object
Step 2: x = x.concat (" Rules!");
x
String reference variable
The heap
"Java"
String object
"Java Rules!"
String object
Notice in step 2 that there is no valid reference to the "Java" String; that object has been "abandoned" and a new object created.

Figure 6-3: An old `String` object being abandoned. The dashed line indicates a deleted reference.

The preceding discussion contains the keys to understanding Java string immutability. If you really, really get the examples and diagrams, backward and forward, you should get 80 percent of the `String` questions on the exam correct.

We will cover more details about strings next, but make no mistake—in terms of bang for your buck, what we've already covered is by far the most important part of understanding how `String` objects work in Java.

We'll finish this section by presenting an example of the kind of devilish `String` question you might expect to see on the exam. Take the time to work it out on paper. (Hint: try to keep track of how many objects and reference variables there are, and which ones refer to which.)

```
String s1 = "spring ";
String s2 = s1 + "summer ";
s1.concat("fall ");
s2.concat(s1);
s1 += "winter ";
System.out.println(s1 + " " + s2);
```

What is the output? For extra credit, how many `String` objects and how many reference variables were created prior to the `println` statement?

Answer: The result of this code fragment is `spring winter spring summer`. There are two reference variables: `s1` and `s2`. A total of eight `String` objects were created as follows: `"spring "`, `"summer "` (lost), `"spring summer "`, `"fall "` (lost), `"spring fall "` (lost), `"spring summer spring "` (lost), `"winter "` (lost), `"spring winter "` (at this point `"spring "` is lost). Only two of the eight `String` objects are not lost in this process.

Important Facts About Strings and Memory

In this section, we'll discuss how Java handles `String` objects in memory and some of the reasons behind these behaviors.

One of the key goals of any good programming language is to make efficient use of memory. As an application grows, it's very common for string literals to occupy large amounts of a program's memory, and there is often a lot of redundancy within the universe of `String` literals for a program. To make Java more memory efficient, the JVM sets aside a special area of memory called the *String constant pool*. When the compiler encounters a `String` literal, it checks the pool to see if an identical `String` already exists. If a match is found, the reference to the new literal is directed to the existing `String`, and no new `String` literal object is created. (The existing `String` simply has an additional reference.) Now you can start to see why making `String` objects immutable is such a good idea. If several reference variables refer to the same `String` without even knowing it, it would be very bad if any of them could change the `String`'s value.

You might say, "Well that's all well and good, but what if someone overrides the `String` class functionality; couldn't that cause problems in the pool?" That's one of the main reasons that the `String` class is marked `final`. Nobody can override the behaviors of any of the `String` methods, so you can rest assured that the `String` objects you are counting on to be immutable will, in fact, be immutable.

Creating New Strings

Earlier we promised to talk more about the subtle differences between the various methods of creating a `String`. Let's look at a couple of examples of how a `String` might be created, and let's further assume that no other `String` objects exist in the pool. In this simple case, `"abc"` will go in the pool, and `s` will refer to it:

```
String s = "abc";     // creates one String object and one
                      // reference variable
```

In the next case, because we used the `new` keyword, Java will create a new `String` object in normal (nonpool) memory, and `s` will refer to it. In addition, the literal `"abc"` will be placed in the pool:

```
String s = new String("abc");      // creates two objects,
                                   // and one reference variable
```

Important Methods in the String Class

The following methods are some of the more commonly used methods in the `String` class, and they are also the ones you're most likely to encounter on the exam.

- **`charAt()`** Returns the character located at the specified index
- **`concat()`** Appends one string to the end of another (+ also works)
- **`equalsIgnoreCase()`** Determines the equality of two strings, ignoring case
- **`length()`** Returns the number of characters in a string
- **`replace()`** Replaces occurrences of a character with a new character
- **`substring()`** Returns a part of a string
- **`toLowerCase()`** Returns a string, with uppercase characters converted to lowercase
- **`toString()`** Returns the value of a string
- **`toUpperCase()`** Returns a string, with lowercase characters converted to uppercase
- **`trim()`** Removes whitespace from both ends of a string

Let's look at these methods in more detail.

public char charAt(int index)

This method returns the character located at the `String`'s specified index. Remember, `String` indexes are zero-based—here's an example:

```
String x = "airplane";
System.out.println( x.charAt(2) );         // output is 'r'
```

public String concat(String s)

This method returns a string with the value of the `String` passed in to the method appended to the end of the `String` used to invoke the method—here's an example:

```
String x = "taxi";
System.out.println( x.concat(" cab") );    // output is "taxi cab"
```

The overloaded + and += operators perform functions similar to the `concat()` method—here's an example:

```
String x = "library";
System.out.println( x + " card");        // output is "library card"

String x = "Atlantic";
x+= " ocean";
System.out.println( x );                 // output is "Atlantic ocean"
```

In the preceding `"Atlantic ocean"` example, notice that the value of `x` really did change! Remember the += operator is an assignment operator, so line 2 is really creating a new string, `"Atlantic ocean"`, and assigning it to the `x` variable. After line 2 executes, the original string `x` was referring to, `"Atlantic"`, is abandoned.

public boolean equalsIgnoreCase(String s)

This method returns a boolean value (`true` or `false`) depending on whether the value of the `String` in the argument is the same as the value of the `String` used to invoke the method. This method will return `true` even when characters in the `String` objects being compared have differing cases—here's an example:

```
String x = "Exit";
System.out.println( x.equalsIgnoreCase("EXIT"));    // is "true"
System.out.println( x.equalsIgnoreCase("tixe"));    // is "false"
```

public int length()

This method returns the length of the `String` used to invoke the method—here's an example:

```
String x = "01234567";
System.out.println( x.length() );      // returns "8"
```

Exam Watch

Arrays have an attribute (not a method) called `length.` You may encounter questions in the exam that attempt to use the `length()` method on an array or that attempt to use the `length` attribute on a `String`. Both cause compiler errors–consider these, for example:

```
String x = "test";
System.out.println( x.length );      // compiler error
```

and

```
String[] x = new String[3];
System.out.println( x.length() );    // compiler error
```

public String replace(char old, char new)

This method returns a `String` whose value is that of the `String` used to invoke the method, but updated so that any occurrence of the char in the first argument is replaced by the char in the second argument—here's an example:

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') );      // output is "oXoXoXoX"
```

public String substring(int begin) and public String substring(int begin, int end)

The `substring()` method is used to return a part (or substring) of the `String` used to invoke the method. The first argument represents the starting location (zero-based) of the substring. If the call has only one argument, the substring returned will include the characters at the end of the original `String`. If the call has two arguments, the substring returned will end with the character located in the *n*th position of the original `String` where *n* is the second argument. Unfortunately, the ending argument is not zero-based, so if the second argument is 7, the last character in the returned `String` will be in the original `String`'s 7 position, which is index 6 (ouch). Let's look at some examples:

```
String x = "0123456789";                    // as if by magic, the value of
each
                                            // char is the same as its
index!
System.out.println( x.substring(5) );       // output is "56789"
System.out.println( x.substring(5, 8));     // output is "567"
```

The first example should be easy: start at index 5 and return the rest of the `String`. The second example should be read as follows: start at index 5 and return the characters up to and including the 8th position (index 7).

public String toLowerCase()

Converts all characters of a `String` to lowercase—here's an example:

```
String x = "A New Moon";
System.out.println( x.toLowerCase() );      // output is "a new moon"
```

public String toString()

This method returns the value of the `String` used to invoke the method. What? Why would you need such a seemingly "do nothing" method? All objects in Java must have a `toString()` method, which typically returns a `String` that in some meaningful way describes the object in question. In the case of a `String` object, what's a more meaningful way than the `String`'s value? For the sake of consistency, here's an example:

```
String x = "big surprise";
System.out.println( x.toString() );        // output? [reader's exercise :-) ]
```

public String toUpperCase()

Converts all characters of a `String` to uppercase–here's an example:

```
String x = "A New Moon";
System.out.println( x.toUpperCase() );     // output is "A NEW MOON"
```

public String trim()

This method returns a `String` whose value is the `String` used to invoke the method, but with any leading or trailing whitespace removed—here's an example:

```
String x = " hi ";
System.out.println( x + "t" );              // output is " hi t"
System.out.println( x.trim() + "t");        // output is "hit"
```

The StringBuilder Class

The `java.lang.StringBuilder` class should be used when you have to make a lot of modifications to strings of characters. As discussed in the previous section, `String` objects are immutable, so if you choose to do a lot of manipulations with `String` objects, you will end up with a lot of abandoned `String` objects in the `String` pool. (Even in these days of gigabytes of RAM, it's not a good idea to waste precious memory on discarded `String` pool objects.) On the other hand, objects of type `StringBuilder` can be modified over and over again without leaving behind a great effluence of discarded `String` objects.

on the job A common use for `StringBuilders` is file I/O when large, ever-changing streams of input are being handled by the program. In these cases, large blocks of characters are handled as units, and `StringBuilder` objects are the ideal way to handle a block of data, pass it on, and then reuse the same memory to handle the next block of data.

Prefer StringBuilder to StringBuffer

The `StringBuilder` class was added in Java 5. It has exactly the same API as the `StringBuffer` class, except `StringBuilder` is not thread-safe. In other words, its methods are not synchronized. Oracle recommends that you use `StringBuilder` instead of `StringBuffer` whenever possible, because `StringBuilder` will run faster (and perhaps jump higher). So apart from synchronization, anything we say about `StringBuilder`'s methods holds true for `StringBuffer`'s methods, and vice versa. That said, for the OCA 8 exam, `StringBuffer` is not tested.

Using StringBuilder (and This Is the Last Time We'll Say This: StringBuffer)

In the previous section, you saw how the exam might test your understanding of `String` immutability with code fragments like this:

```
String x = "abc";
x.concat("def");
System.out.println("x = " + x);       // output is "x = abc"
```

Because no new assignment was made, the new `String` object created with the `concat()` method was abandoned instantly. You also saw examples like this:

```
String x = "abc";
x = x.concat("def");
System.out.println("x = " + x);       // output is "x = abcdef"
```

We got a nice new `String` out of the deal, but the downside is that the old `String "abc"` has been lost in the `String` pool, thus wasting memory. If we were using a `StringBuilder` instead of a `String`, the code would look like this:

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def");
System.out.println("sb = " + sb);     // output is "sb = abcdef"
```

All of the `StringBuilder` methods we will discuss operate on the value of the `StringBuilder` object invoking the method. So a call to `sb.append("def");` is actually appending `"def"` to itself (`StringBuilder sb`). In fact, these method calls can be chained to each other—here's an example:

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb );              // output is "fed---cba"
```

Notice that in each of the previous two examples, there was a single call to `new`, so in each example we weren't creating any extra objects. Each example needed only a single `StringBuilder` object to execute.

Exam Watch

So far we've seen `StringBuilders` being built with an argument specifying an initial value. `StringBuilders` can also be built empty, and they can also be constructed with a specific size or, more formally, a "capacity." For the exam, there are three ways to create a new `StringBuilder`:

```
1. new StringBuilder();        // default cap. = 16 chars
2. new StringBuilder("ab");    // cap. = 16 + arg's length
3. new StringBuilder(x);       // capacity = x (an integer)
```

The two most common ways to work with `StringBuilders` is via an `append()` method or an `insert()` method. In terms of a

StringBuilder's capacity, there are three rules to keep in mind when appending and inserting:

If an append() grows a StringBuilder past its capacity, the capacity is updated automatically.

If an insert() starts within a StringBuilder's capacity but ends after the current capacity, the capacity is updated automatically.

If an insert() attempts to start at an index after the StringBuilder's current length, an exception will be thrown.

Important Methods in the StringBuilder Class

The StringBuilder class has a zillion methods. Following are the methods you're most likely to use in the real world and, happily, the ones you're most likely to find on the exam.

public StringBuilder append(String s)

As you've seen earlier, this method will update the value of the object that invoked the method, whether or not the returned value is assigned to a variable. Versions of this heavily overloaded method will take many different arguments, including boolean, char, double, float, int, long, and others, but the one most likely used on the exam will be a String argument—for example,

```
StringBuilder sb = new StringBuilder("set ");
sb.append("point");
System.out.println(sb);       // output is "set point"
StringBuilder sb2 = new StringBuilder("pi = ");
sb2.append(3.14159f);
System.out.println(sb2);      // output is "pi = 3.14159"
```

public StringBuilder delete(int start, int end)

This method modifies the value of the StringBuilder object used to invoke it. The starting index of the substring to be removed is defined by the first argument (which is zero-based), and the ending index of the substring to be removed is defined by the second argument (but it is one-based)! Study the following example carefully:

```
StringBuilder sb = new StringBuilder("0123456789");
System.out.println(sb.delete(4,6));      // output is "01236789"
```

Exam Watch

The exam will probably test your knowledge of the difference between String and StringBuilder objects. Because StringBuilder objects are changeable, the following code fragment will behave differently than a similar code fragment that uses String objects:

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def");
System.out.println( sb );
```

In this case, the output will be: "abcdef"

---

public StringBuilder insert(int offset, String s)

This method updates the value of the StringBuilder object that invoked the method call. The String passed in to the second argument is inserted into the StringBuilder starting at the offset location represented by the first argument (the offset is zero-based). Again, other types of data can be passed in through the second argument (**boolean, char, double, float, int, long**, and so on), but the String argument is the one you're most likely to see:

```
StringBuilder sb = new StringBuilder("01234567");
sb.insert(4, "---");
System.out.println( sb );         // output is "0123---4567"
```

public StringBuilder reverse()

This method updates the value of the StringBuilder object that invoked the method call. When invoked, the characters in the StringBuilder are reversed—the first character becoming the last, the second becoming the second to the last, and so on:

```
StringBuilder sb = new StringBuilder("A man a plan a canal Panama");
sb.reverse();
System.out.println(sb); // output: "amanaP lanac a nalp a nam A"
```

public String toString()

This method returns the value of the StringBuilder object that invoked the method call as a String:

```
StringBuilder sb = new StringBuilder("test string");
System.out.println( sb.toString() );  // output is "test string"
```

That's it for `StringBuilders`. If you take only one thing away from this section, it's that unlike `String` objects, `StringBuilder` objects can be changed.

Exam Watch

Many of the exam questions covering this chapter's topics use a tricky bit of Java syntax known as "chained methods." A statement with chained methods has this general form:

```
result = method1().method2().method3();
```

In theory, any number of methods can be chained in this fashion, although typically you won't see more than three. Here's how to decipher these "handy Java shortcuts" when you encounter them:

1. Determine what the leftmost method call will return (let's call it x).
2. Use x as the object invoking the second (from the left) method. If there are only two chained methods, the result of the second method call is the expression's result.
3. If there is a third method, the result of the second method call is used to invoke the third method, whose result is the expression's result–for example,

```
String x = "abc";
String y = x.concat("def").toUpperCase().replace('C','x'); //chained methods
System.out.println("y = " + y); // result is "y = ABxDEF"
```

Let's look at what happened. The literal `def` was concatenated to `abc`, creating a temporary, intermediate `String` (soon to be lost), with the value `abcdef`. The `toUpperCase()` method was called on this `String,` which created a new (soon to be lost) temporary `String` with the value `ABCDEF.` The `replace()` method was then called on this second `String` object, which created a final `String` with the value `ABxDEF` and referred y to it.

CERTIFICATION OBJECTIVE: WORKING WITH CALENDAR DATA (OCA OBJECTIVE 9.3)

9.3 Create and manipulate calendar data using the following classes: java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period

Java 8 introduced a large collection (argh) of new packages related to working with calendars, dates, and times. The OCA 8 creators chose to include knowledge of a subset of these packages and classes as an exam objective. If you understand the classes included in the exam objective, you'll have a good introduction to the entire calendar/date/time topic. As we work through this section, we'll use the phrase "calendar object," which we use to refer to objects of one of the several types of calendar-related classes we're covering. So "calendar object" is a made-up umbrella term. Here's a summary of the five calendar-related classes we'll study, plus an interface that looms large:

- **`java.time.LocalDateTime`** This class is used to create immutable objects, each of which represents a specific date and time. Additionally, this class provides methods that can manipulate the values of the date/time objects created and assign them to new immutable objects. `LocalDateTime` objects contain BOTH information about days, months, and years, AND about hours, minutes, seconds, and fractions of seconds.
- **`java.time.LocalDate`** This class is used to create immutable objects, each of which represents a specific date. Additionally, this class provides methods that can manipulate the values of the date objects created and assign them to new immutable objects. `LocalDate` objects are accurate only to days. Hours, minutes, and seconds are **not** part of a `LocalDate` object.
- **`java.time.LocalTime`** This class is used to create immutable objects, each of which represents a specific time. Additionally, this class provides methods that can manipulate the values of the time objects created and assign them to new immutable objects. `LocalTime` objects refer only to hours, minutes, seconds, and fractions of seconds. Days, months, and years are **not** a part of `LocalTime` objects.
- **`java.time.format.DateTimeFormatter`** This class is used by the classes just described to format date/time objects for output and to parse input strings and convert them to date/time objects. `DateTimeFormatter` objects are also immutable.
- **`java.time.Period`** This class is used to create immutable objects that represent a period of time, for example, "one year, two months, and three days." This class works in years, months, and days. If you want to represent chunks of time in increments finer than a day (e.g., hours and minutes), you can use the `java.time.Duration` class, but `Duration` is not on the exam.
- **`java.time.temporal.TemporalAmount`** This interface is implemented by the `Period` class. When you use `Period` objects to manipulate (see the following section), calendar objects, you'll often use methods that take objects that implement `TemporalAmount`. In general, as you use the Java API more and more, it's a good idea to learn which classes implement which interfaces; this is a key way to learn how the classes in complex packages interact with each other.

Immutability

There are a couple of recurring themes in the previous definitions. First, notice that most of the calendar-related objects you'll create are **immutable**. Just like `String` objects! So when we say we're going to "manipulate" a calendar object, what we "really" mean is that we'll invoke a method on a calendar object, and we'll return a new calendar object that represents the result of **manipulating the value** of the original calendar object. But the original calendar object's value is not, and cannot, be changed. Just like `Strings`! Let's see an example:

```
LocalDate date1 = LocalDate.of(2017, 1, 31);
    Period period1 = Period.ofMonths(1);
    System.out.println(date1);
    date1.plus(period1);                    // new value is lost
    System.out.println(date1);
    LocalDate date2 = date1.plus(period1);  // new value is captured
    System.out.println(date2);
```

which produces:

```
2017-01-31
2017-01-31
2017-02-28
```

Notice that invoking the `plus` method on `date1` doesn't change its value, but assigning the result of the `plus` method to `date2` captures a new value. Expect exam questions that test your understanding of the immutability of calendar objects.

Factory Classes

The next thing to notice in the previous code listing is that we never used the keyword `new` in the code. We didn't directly invoke a constructor. None of the five classes listed in OCA 8 objective 9.3 have public constructors. Instead, for all these classes, you invoke a `public static` method in the class to create a new object. As you go further into your studies of OO design, you'll come across the phrases "factory pattern," "factory methods," and "factory classes." Usually, when a class has no public constructors and provides at least one `public static` method that can create new instances of the class, that class is called a *factory class*, and any method that is invoked to get a new instance of the class is called a *factory method*. There are many good reasons to create factory classes, most of which are beyond the scope of this book, but one of them we will discuss now. If we use the `LocalDate` class as an example, we find the following `static` methods that create and return a new instance:

```
from()
now()           // three overloaded methods exist
of()            // two overloaded methods exist
ofEpochDay()
ofYearDay()
parse()         // two overloaded methods exist
```

So we have what, about ten different ways to create a new `LocalDate` object? By using methods with different names (instead of using overloaded constructors), the method names themselves make the code more readable. It's clearer what variation of `LocalDate` we're making. As you use more and more classes from the Java API, you'll discover that the API creators use factory classes a lot.

Exam Watch

Whenever you see an exam question relating to dates or times, be on the lookout for the `new` keyword. This is your tipoff that the code won't compile:

```
LocalDateTime d1 = new LocalDateTime(); // won't compile
```

Remember the exam's date and time classes use factory methods to create new objects.

Using and Manipulating Dates and Times

Now that we know how to create new calendar-related objects, let's turn to using and manipulating them. (And you know what we mean when we say "manipulate.") The following code demonstrates some common uses and powerful features of the new Java 8 calendar-related classes:

```
import java.time.*;
import java.time.format.*;
import java.time.temporal.ChronoUnit;                       // not on the exam
                                                            // but VERY useful
public class DrWho {
  public static void main(String[] args) {
    DateTimeFormatter f =
              DateTimeFormatter.ofPattern("MMddyyyy"); // describe a format
    LocalDate bday = null;
    try {
      bday = LocalDate.parse(args[0], f);              // verify input date
                                                       // often parse() methods
                                                       // throw exceptions!
    } catch (java.time.DateTimeException e) {
      System.out.println("bad dates Indy");
      System.exit(0);
    }
    System.out.println("your birthday is: " + bday);
    System.out.println("a " + bday.getDayOfWeek());    // useful

    Period p1 = Period.between(bday, LocalDate.now()); // very useful!

    System.out.println("you've lived for: ");
    System.out.print(p1.getDays() + " days, ");        // split up a Period
    System.out.print(p1.getMonths() + " months, ");
    System.out.println(p1.getYears() + " years");

    int yearsOld = p1.getYears();
    if(yearsOld < 0 || yearsOld > 119)
      System.out.println("Wow, are you a time lord?");

    long tDays = bday.until(LocalDate.now(),           // handy method +
                            ChronoUnit.DAYS);          // handy enum
                                                       // = powerful date math

    System.out.println("you've lived for " + tDays
                        + " days, so far");

    System.out.println("you'll reach 30,000 days on "
                        + bday.plusDays(30_000));      // date math

    LocalDate d2000 = LocalDate.of(2_000, 1, 1);       // of() is a
                                                       // commonly used
                                                       // 'factory' method

    Period p2 = Period.between(d2000, LocalDate.now());
    System.out.println("period since Y2K: " + p2);
  }
}
```

Invoking the program with a relevant birthday:

```
java 01201934
```

produces the output (when run on January 13, 2017):

```
your birthday is: 1934-01-20
a SATURDAY
you've lived for:
24 days, 11 months, 82 years
 you've lived for 30309 days, so far
 you'll reach 30000 days on 2016-03-10
 period since Y2K: P17Y12D
```

There's a lot going on here, so let's do a walk-through. First, we want users to enter their birthday in the form of *mmddyyyy*. We use a `DateTimeFormatter` object to parse the user's first argument and verify that it's a valid date of the form we're hoping for. Usually in the Java API, `parse()` methods can throw exceptions, so we have to do our parsing in a `try/catch` block.

Next, we print out the verified date and show off a bit by printing out what day of the week that date occurred on. This calculation would be quite tricky to do by hand!

Next, we create a `Period` object that represents the amount of time between the user's birthday and today, and we use various `getX()` methods to list the details of the `Period` object.

After making sure we're not dealing with a time lord, we then use the very powerful `until()` method and "day" as the unit of time to determine how many days the user has been alive. We cheated a bit here and used the `ChronoUnit` enum from the `java.time.temporal` package. (Even though `ChronoUnit` isn't on the exam, we think if you do a lot of calendar calculations, you'll end up using this enum a lot.)

Next, we add 30,000 days to the user's birthday so we can calculate on which date our user will have lived for 30,000 days. It's a short jump to seeing how these sorts of calendar calculations will be very powerful for scheduling applications, project management applications, travel planning, and so on.

Finally, we use a common factory method, `of()`, to create another date object (representing today's date), and we use that in conjunction with the very powerful `between()` method to see how long it's been since January 1, 2000, Y2K.

Formatting Dates and Times

Now let's turn to formatting dates and times using the `DateTimeFormatter` class, so your calendar objects will look all shiny when you want to include them in your program's output. For the exam, you should know the following two-step process for creating `Strings` that represent well-formatted calendar objects:

1. Use formatters and patterns from the HUGE lists provided in the `DateTimeFormatter` class to create a `DataTimeFormatter` object.
2. In the `LocalDate`, `LocalDateTime`, and `LocalTime` classes, use the `format()` method with the `DateTimeFormatter` object as the argument to create a well-formed `String`—or use the `DateTimeFormatter.format()` method with a calendar argument to create a well-formed `String`. Let's look at a few examples:

```
import java.time.*;
import java.time.format.*;
public class NiceDates {
  public static void main(String[] args) {
    DateTimeFormatter f1 =
              DateTimeFormatter.ofPattern("MMM dd, yyyy");
    DateTimeFormatter f2 =
              DateTimeFormatter.ofPattern("E MMM dd, yyyy G");
    DateTimeFormatter tf1 =
              DateTimeFormatter.ofPattern("k:m:s A a");

    LocalDate d = LocalDate.now();
    String s = d.format(f1);           // thus proving that the format()
                                       // method makes String objects

    System.out.println(s);
    System.out.println(d.format(f2));

    LocalTime t = LocalTime.now();
    System.out.println(t.format(tf1));
  }
}
```

which, when we ran this code, produced the following (your output will vary):

```
Jan 14, 2017
Sat Jan 14, 2017 AD
14:17:9 51429958 PM
```

Some of the pattern codes we used are self-evident (e.g. MMM dd yyyy), and some are fairly arbitrary like "E" for day of week or "k" for military hours. All of the codes can be found in the `DateTimeFormatter` API.

That's enough about calendars; on to arrays!

CERTIFICATION OBJECTIVE: USING ARRAYS (OCA OBJECTIVES 4.1 AND 4.2)

4.1 Declare, instantiate, initialize, and use a one-dimensional array.

4.2 Declare, instantiate, initialize, and use a multi-dimensional array.

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 these, and you need to know about all of them for the exam.

on the job Arrays are efficient, but most of the time you'll want to use one of the Collection types from java.util (including `HashMap`, `ArrayList`, and `TreeSet`). Collection classes offer more flexible ways to access an object (for insertion, deletion, and so on), and unlike arrays, they 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? The OCP 8 exam covers collections 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 in the exam that include code similar to the following:

```
int[5] scores; // will NOT compile
```

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)—that is, 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
                            // to 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 6-4 shows the `testScores` array on the heap, after construction.

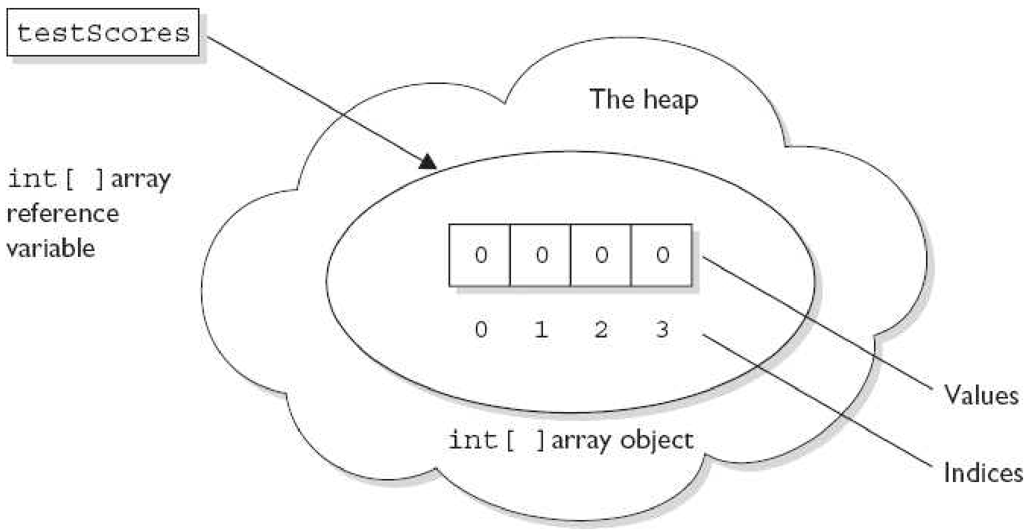


Figure 6-4: A one-dimensional array on the heap

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]; // no Thread objects created!
                                  // one Thread array created
```

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 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

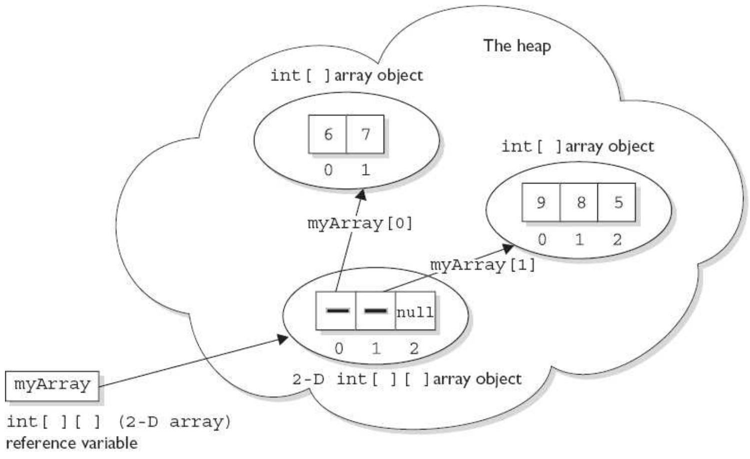
Multidimensional arrays, remember, are simply arrays of arrays. So a two-dimensional array of type `int` is really an object of type `int` array (`int []`), with each element in that array holding a reference to another `int` array. The second dimension holds the actual `int` primitives.

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 because the JVM needs to know only the size of the object assigned to the variable `myArray`.

Figure 6-5 shows how a two-dimensional `int` array works 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;
```

Figure 6-5: A two-dimensional array on the heap

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 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 actually try to 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 (0), 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 element `[3]` will raise an `ArrayIndexOutOfBoundsException,` because in an array of three elements, the legal index values are 0, 1, and 2. You also might see an attempt to use a negative number as an array index. The following are examples of legal and illegal array access attempts. Be sure to recognize that these cause runtime exceptions and not compiler errors!

Nearly all 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
```

numberThese 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 (scores) 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 is an int array of six int elements

scores[2] = new int[1];
// the third element 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 index position x
}
```

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

Declaring, Constructing, and Initializing on One Line

You can use two different array-specific syntax shortcuts both to 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:

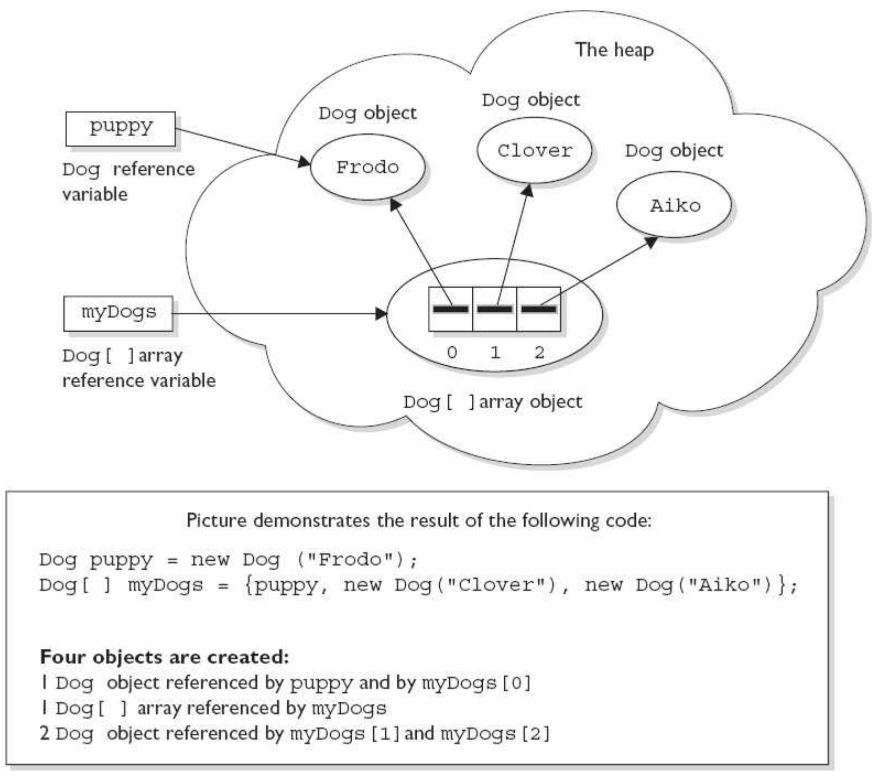
```
int[] dots;
dots = new int[3];
int x = 9;
dots[0] = 6;
dots[1] = x;
dots[2] = 8;
```

This begs the question, "Why would anyone use the longer way?" One reason comes to mind. You might not know—at the time you create the array—the values that will be assigned to the array's elements.

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. This array shortcut alone (combined with the stimulating prose) is worth the price of this book. Figure 6-6 shows the result.



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 object referenced by myDogs[1] and myDogs[2]

Figure 6-6: Declaring, constructing, and initializing an array of objects

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

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

This 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 comma-separated items 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 `5,2,4,7`. The following code shows the values of some of the elements in this two-dimensional array:

```
scores[0]     // an array of 4 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
```

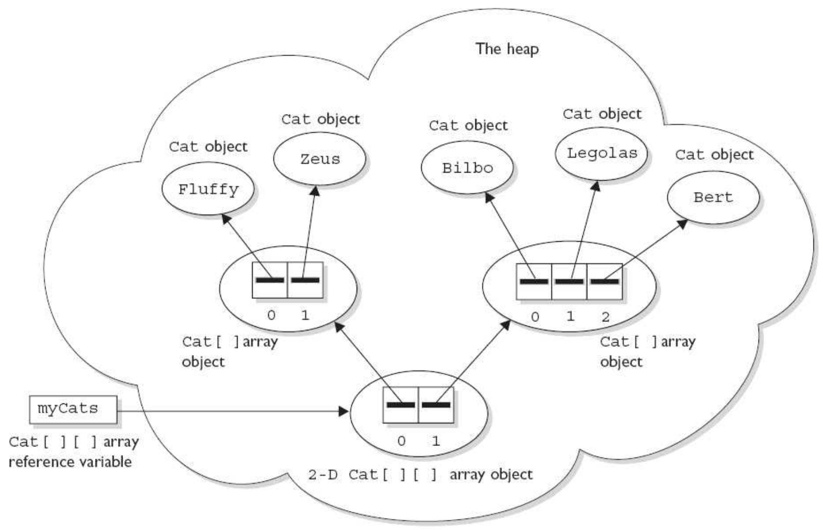
Figure 6-7 shows the result of declaring, constructing, and initializing a two-dimensional array in one statement.

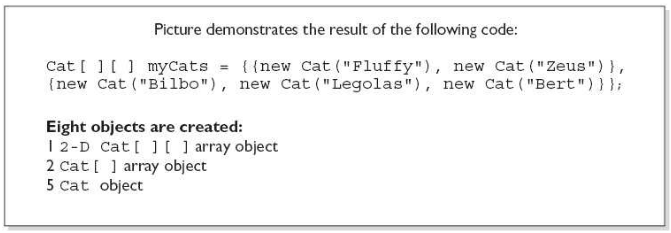
Constructing and Initializing an Anonymous Array

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

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

The preceding code creates a new `int` array with three elements; initializes the three elements with the values `4`, `7`, and `2`; and then assigns the new array to the previously declared `int` array reference variable `testScores`.




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 object
5 Cat object

Figure 6-7: Declaring, constructing, and initializing a two-dimensional array

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 JIT {
  void takesAnArray(int[] someArray) { // use the array
  public static void main (String [] args) {
    JIT j = new JIT();
    j.takesAnArray(new int[] {7,7,8,2,5}); // pass an array
}}
```

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 this:

```
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 RacingFlat-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··

        // 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 the variable 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).

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
```

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.

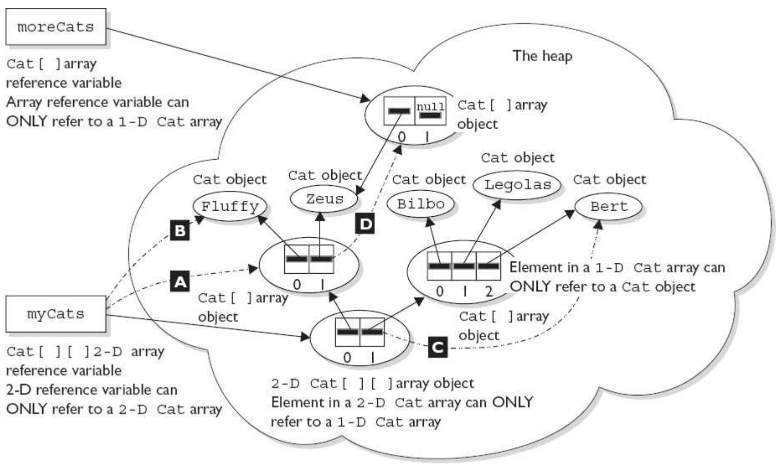
Array Reference Assignments for Multidimensional Arrays When you assign an array to a previously declared array reference, the array you're assigning must be in the same dimension as the reference you're assigning it to. For example, a two-dimensional array of `int` arrays cannot be assigned to a regular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

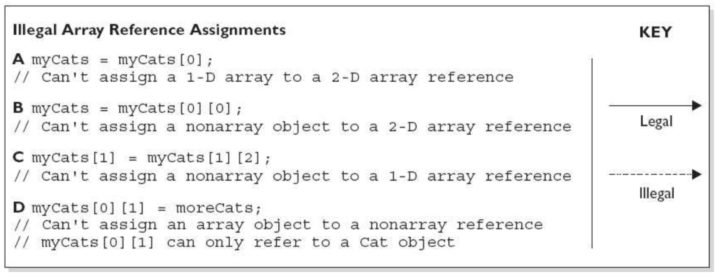
```
int[] blots;
int[][] squeegees = new int[3][];
blots = squeegees;         // NOT OK, squeegees is a
                           // two-d array of int arrays
int[] blocks = new int[6];
blots = blocks;            // OK, blocks is an int array
```

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 6-8 shows an example of legal and illegal assignments for references to an array.


moreCats
Cat[ ] array reference variable
Array reference variable can ONLY refer to a 1-D Cat array
The heap
null
0 1
Cat[ ] array object
Cat object
Fluffy
Cat object
Zeus
Cat object
Bilbo
Cat object
Legolas
Cat object
Bert
B
D
A
0 1
Cat[ ] array object
0 1 2
Element in a 1-D Cat array can ONLY refer to a Cat object
myCats
Cat[ ][ ] 2-D array reference variable
2-D reference variable can ONLY refer to a 2-D Cat array
0 1
Cat[ ] array object
C
2-D Cat[ ][ ] array object
Element in a 2-D Cat array can ONLY refer to a 1-D Cat array


Illegal Array Reference Assignments

KEY

Legal

Illegal

```
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
```

Figure 6-8: Legal and illegal array assignments

CERTIFICATION OBJECTIVE: USING ARRAYLISTS AND WRAPPERS (OCA OBJECTIVES 9.4 AND 2.5)

9.3 Declare and use an ArrayList of a given type.

2.5 Develop code that uses wrapper classes such as Boolean, Double, and Integer.

Data structures are a part of almost every application you'll ever work on. The Java API provides an extensive range of classes that support common data structures such as `Lists`, `Sets`, `Maps`, *and* `Queues`. For the purpose of the OCA exam, you should remember that the classes that support these common data structures are a part of what is known as "The Collection API" (one of its many aliases). (The OCP exam covers the most common implementations of all these structures.)

When to Use ArrayLists

We've already talked about arrays. Arrays seem useful and pretty darned flexible. So why do we need more functionality than arrays provide? Consider these two situations:

- You need to be able to increase and decrease the size of your list of things.
- The order of things in your list is important and might change.

Both situations can be handled with arrays, but it's not easy….

Suppose you want to plan a vacation to Europe. You have several destinations in mind (Paris, Oslo, Rome), but you're not yet sure in what order you want to visit these cities, and as your planning progresses, you might want to add or subtract cities from your list. Let's say your first idea is to travel from north to south, so your list looks like this:

Oslo, Paris, Rome.

If we were using an array, we could start with this:

```
String[] cities = {"Oslo", "Paris", "Rome"};
```

But now imagine that you remember that you REALLY want to go to London, too! You've got two problems:

- Your cities array is already full.
- If you're going from north to south, you need to insert London before Paris.

Of course, you can figure out a way to do this. Maybe you create a second array, and you copy cities from one array to the other, and at the correct moment you add London to the second array. Doable, but difficult.

Now let's see how you could do the same thing with an `ArrayList`:

```
import java.util.*;                               // ArrayList lives in .util
public class Cities {
  public static void main(String[] args) {

    List<String> c = new ArrayList<String>();     // create an ArrayList, c
    c.add("Oslo");                                // add original cities
    c.add("Paris");
    c.add("Rome");
    int index = c.indexOf("Paris");               // find Paris' index
    System.out.println(c + " " + index);
    c.add(index, "London");                       // add London before Paris
    System.out.println(c);                        // show the contents of c
  }
}
```

The output will be something like this:

```
[Oslo, Paris, Rome] 1
[Oslo, London, Paris, Rome]
```

By reviewing the code, we can learn some important facts about `ArrayLists`:

- The `ArrayList` class is in the `java.util` package.
- Similar to arrays, when you build an `ArrayList`, you have to declare what kind of objects it can contain. In this case, we're building an `ArrayList` of `String` objects. (We'll look at the line of code that creates the `ArrayList` in a lot more detail in a minute.)
- `ArrayList` implements the List interface.
- We work with the `ArrayList` through methods. In this case we used a couple of versions of `add()`; we used `indexOf()`; and, indirectly, we used `toString()` to display the `ArrayList`'s contents. (More on `toString()` in a minute.)
- Like arrays, indexes for `ArrayLists` are zero-based.

- We didn't declare how big the `ArrayList` was when we built it.
- We were able to add a new element to the `ArrayList` on the fly.
- We were able to add the new element in the middle of the list.
- The `ArrayList` maintained its order.

As promised, we need to look at the following line of code more closely:

```
List<String> c = new ArrayList<String>();
```

First off, we see that this is a polymorphic declaration. As we said earlier, `ArrayList` implements the `List` interface (also in `java.util`). If you plan to take the OCP 8 exam after you've aced the OCA 8, you'll learn a lot more about why we might want to do a polymorphic declaration. For now, imagine that someday you might want to create a `List` of your `ArrayLists`.

Next, we have this weird-looking syntax with the < and > characters. This syntax was added to the language in Java 5, and it has to do with "generics." Generics aren't really included in the OCA exam, so we don't want to spend a lot of time on them here, but what's important to know is that this is how you tell the compiler and the JVM that for this particular `ArrayList` you want only `Strings` to be allowed. What this means is if the compiler can tell that you're trying to add a "not-a-`String`" object to this `ArrayList`, your code won't compile. This is a good thing!

Also as promised, let's look at THIS line of code:

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

Remember that all classes ultimately inherit from class `Object`. Class `Object` contains a method called `toString()`. Again, `toString()` isn't "officially" on the OCA exam (of course, it IS in the OCP exam!), but you need to understand it a bit for now. When you pass an object reference to either `System.out.print()` or `System.out.println()`, you're telling them to invoke that object's `toString()` method. (Whenever you make a new class, you can optionally override the `toString()` method your class inherited from `Object` to show useful information about your class's objects.) The API developers were nice enough to override `ArrayList`'s `toString()` method for you to show the contents of the `ArrayList`, as you saw in the program's output. Hooray!

ArrayLists and Duplicates

As you're planning your trip to Europe, you realize that halfway through your stay in Rome, there's going to be a fantastic music festival in Naples! Naples is just down the coast from Rome! You've got to add that side trip to your itinerary. The question is, can an `ArrayList` have duplicate entries? Is it legal to say this:

```
c.add("Rome");
c.add("Naples");
c.add("Rome");
```

And the short answer is: **Yes, ArrayLists can have duplicates**. Now if you stop and think about it, the notion of "duplicate Java objects" is actually a bit tricky. Relax, because you won't have to get into that trickiness until you study for the OCP 8.

Exam Watch

Technically speaking, `ArrayLists` hold only object references, not actual objects and not primitives. If you see code like this,

```
myArrayList.add(7);
```

what's really happening is the `int` is being autoboxed (converted) into an `Integer` object and then added to the `ArrayList`. We'll talk more about autoboxing in a few pages.

ArrayList Methods in Action

Let's look at another piece of code that shows off most of the `ArrayList` methods you need to know for the exam:

```
import java.util.*;
public class TweakLists {
  public static void main(String[] args) {

    List<String> myList = new ArrayList<String>();

    myList.add("z");
    myList.add("x");
    myList.add(1, "y");              // zero based
    myList.add(0, "w");              // " "
    System.out.println(myList);      // [w, z, y, x]
    myList.clear();                  // remove everything
    myList.add("b");
    myList.add("a");
    myList.add("c");
    System.out.println(myList);     // [b, a, c]
    System.out.println(myList.contains("a") + " " + myList.contains("x"));

    System.out.println("get 1: " + myList.get(1));
    System.out.println("index of c: " + myList.indexOf("c"));

    myList.remove(1);               // remove "a"
    System.out.println("size: " + myList.size() + " contents: " + myList);
 }
}
```

which should produce something like this:

```
[w, z, y, x]
[b, a, c]
true false
get 1: a
index of c: 2
size: 2 contents: [b, c]
```

A couple of quick notes about this code: First off, notice that `contains()` returns a boolean. This makes `contains()` great to use in "if" tests. Second, notice that `ArrayList` has a `size()` method. It's important to remember that arrays have a length attribute and `ArrayLists` have a `size()` method.

Important Methods in the ArrayList Class

The following methods are some of the more commonly used methods in the `ArrayList` class and also those that you're most likely to encounter on the exam:

- **`add(element)`** Adds this element to the **end** of the `ArrayList`
- **`add(index, element)`** Adds this element at the index point and shifts the remaining elements back (for example, what was at `index` is now at `index + 1`)
- **`clear()`** Removes all the elements from the `ArrayList`
- **`boolean contains(element)`** Returns whether the `element` is in the list
- **`Object get(index)`** Returns the `Object` located at `index`
- **`int indexOf(Object)`** Returns the (`int`) location of the element or `-1` if the `Object` is not found
- **`remove(index)`** Removes the element at that `index` and shifts later elements toward the beginning one space
- **`remove(Object)`** Removes the **first** occurrence of the `Object` and shifts later elements toward the beginning one space
- **`int size()`** Returns the number of elements in the `ArrayList`

To summarize, the OCA 8 exam tests only for very basic knowledge of `ArrayLists`. If you go on to take the OCP 8 exam, you'll learn a lot more about `ArrayLists` and other common collections-oriented classes.

Autoboxing with ArrayLists

In general, collections like `ArrayList` can hold objects but not primitives. Prior to Java 5, a common use for the so-called wrapper classes (e.g., `Integer`, `Float`, `Boolean`, and so on) was to provide a way to get primitives into and out of collections. Prior to Java 5, you had to "wrap" a primitive manually before you could put it into a collection. As of Java 5, primitives still have to be wrapped before they can be added to ArrayLists, but autoboxing takes care of it for you.

```
List myInts = new ArrayList();   // pre Java 5 declaration
myInts.add(new Integer(42));     // Use Integer class to "wrap" an int
```

In the previous example, we create an instance of class `Integer` with a value of `42`. We've created an entire object to "wrap around" a primitive value. As of Java 5, we can say:

```
myInts.add(42); // autoboxing handles it!
```

In this last example, we are still adding an `Integer` object to `myInts` (not an `int` primitive); it's just that autoboxing handles the wrapping for us. There are some sneaky implications when we need to use wrapper objects; let's take a closer look…

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);             // rewrap it
System.out.println("y = " + y); // print it
```

Now you can say:

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

Both examples produce the following output:

```
y = 568
```

And yes, you read that correctly. The code appears to be using the postincrement 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, however, 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 `y++;` it had to substitute something like this:

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

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

Exam Watch

All the wrapper classes except `Character` provide two constructors: one takes a primitive of the type being constructed, and the other takes a `String` representation of the type being constructed. For example,

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

are both valid ways to construct a new `Integer` object (that "wraps" the value 42).

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()`. 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 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 (created through boxing) 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`

When == is used to compare a primitive to a wrapper, the wrapper will be unwrapped and the comparison will be primitive to primitive.

Where Boxing Can Be Used

As we discussed earlier, it's 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 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.

The Java 7 "Diamond" Syntax

Earlier in the book, we discussed several small additions/improvements to the language that were added under the name "Project Coin." The last Project Coin improvement we'll discuss is the "diamond syntax." We've already seen several examples of declaring type-safe `ArrayLists` like this:

```
ArrayList<String> stuff = new ArrayList<String>();
ArrayList<Dog> myDogs = new ArrayList<Dog>();
```

Notice that the type parameters are duplicated in these declarations. As of Java 7, these declarations could be simplified to:

```
ArrayList<String> stuff = new ArrayList<>();
ArrayList<Dog> myDogs = new ArrayList<>();
```

Notice that in the simpler Java 7 declarations, the right side of the declaration included the two characters "<>," which together make a diamond shape—doh!

You cannot swap these; for example, the following declaration is NOT legal:

```
ArrayList<> stuff = new ArrayList<String>(); // NOT a legal diamond syntax
```

For the purposes of the exam, that's all you'll need to know about the diamond operator. For the remainder of the book, we'll use the pre-diamond syntax and the Java 7 diamond syntax somewhat randomly—just like the real world!

CERTIFICATION OBJECTIVE: ADVANCED ENCAPSULATION (OCA OBJECTIVE 6.5)

6.5 Apply encapsulation principles to a class.

Encapsulation for Reference Variables

In Chapter 2 we began our discussion of the object-oriented concept of encapsulation. At that point, we limited our discussion to protecting a class's primitive fields and (immutable) `String` fields. Now that you've learned more about what it means to "pass-by-copy" and we've looked at nonprimitive ways of handling data such as arrays, `StringBuilders`, and `ArrayLists`, it's time to take a closer look at encapsulation.

Let's say we have some special data whose value we're saving in a `StringBuilder`. We're happy to share the value with other programmers, but we don't want them to change the value:

```
class Special {
  private StringBuilder s = new StringBuilder("bob");         // our special data
  StringBuilder getName() { return s; }
  void printName() { System.out.println(s); }                 // verify our special
                                                              // data
}
public class TestSpecial {
  public static void main(String[] args) {
    Special sp = new Special();
    StringBuilder s2 = sp.getName();
    s2.append("fred");
    sp.printName();
  }
}
```

When we run the code, we get this:

```
bobfred
```

Uh oh! It looks like we practiced good encapsulation techniques by making our field private and providing a "getter" method, but based on the output, it's clear that we didn't do a very good job of protecting the data in the `Special` class. Can you figure out why? Take a minute….

Okay—just to verify your answer—when we invoke `getName()`, we do, in fact, return a copy, just like Java always does. But we're not returning a copy of the `StringBuilder` object; we're returning a copy of the reference variable that points to (I know) the one and only `StringBuilder` object we ever built. So at the point that `getName()` returns, we have one `StringBuilder` object and two reference variables pointing to it (`s` and `s2`).

For the purpose of the OCA exam, the key point is this: When encapsulating a mutable object like a `StringBuilder`, or an array, or an `ArrayList`, if you want to let outside classes have a copy of the object, you must actually copy the object and return a reference variable to the object that is a copy. If all you do is return a copy of the original object's reference variable, you **DO NOT** have encapsulation.

CERTIFICATION OBJECTIVE: USING SIMPLE LAMBDAS (OCA OBJECTIVE 9.5)

9.5 Write a simple Lambda expression that consumes a Lambda Predicate expression.

Java 8 is probably best known as the version of Java that finally added lambdas and streams. These two new features (lambdas and streams) give programmers tools to tackle some common and complex problems with easier-to-read, more concise, and, in many cases, faster-running code. The creators of the OCA 8 exam felt that, in general, lambdas and streams are topics more appropriate for the OCP 8 exam, but they wanted OCA 8 candidates to get an introduction, perhaps to whet their appetite…

In this section, we're going to do a really basic introduction to lambdas. We suspect this discussion will raise some questions in your mind, and we're sorry for that, but we're going to restrict ourselves to just the introduction that the exam creators had in mind.

Exam Watch

On the real exam, you should expect to see many questions that test for more than one objective. In the following pages, as we discuss lambdas, we'll be leaning heavily on `ArrayLists` and wrapper classes. You'll see that we combine `ArrayLists`, wrappers, AND lambdas into many of our code listings, and we also use this combination in the mock exam questions we provide. The real exam (and real-life programming) will do the same.

Suppose you're creating an application for a veterinary hospital. We want to focus on that part of the application that allows the vets to get summary information about all the dogs that they work with. Here's our `Dog` class:

```
class Dog {
  String name;
  int weight;
  int age;
  // constructor assigns a name, weight and age
  Dog(String name, int weight, int age) {
    this.name = name;
    this.weight = weight;
    this.age = age;
  }
  String getName() {return name;}
  int getWeight() { return weight;}
  int getAge() { return age;}
  public String toString() {
    return name;
  }
}
```

Now let's write some test code to create some sample `Dogs`, put them into an `ArrayList` as we go, and then run some "queries" against the `ArrayList`.

First here's the summary pseudo code:

```
// create and populate an ArrayList of Dog objects
// invoke a few "queries" on the ArrayList
// declare a couple of "query" methods
```

Here's the actual test code:

```
import java.util.*;
public class TestDogs {
  public static void main(String[] args) {
    ArrayList<Dog> dogs = new ArrayList<>();   // create and populate
    dogs.add(new Dog("boi", 30, 6)); dogs.add(new Dog("tyri", 40, 12));
    dogs.add(new Dog("charis", 120, 7)); dogs.add(new Dog("aiko", 50, 10));
    dogs.add(new Dog("clover", 35, 12)); dogs.add(new Dog("mia", 15, 4));
    dogs.add(new Dog("zooey", 45, 8));
                                               // run a few "queries"
    System.out.println("all dogs " + dogs);
    System.out.println("min age 7 " + minAge(dogs, 7).toString());
    System.out.println("max wght. " + maxWeight(dogs,40).toString());
  }
                                               // declare "query" methods
static ArrayList<Dog> minAge(ArrayList<Dog> dogList, int testFor) {
  ArrayList<Dog> result1 = new ArrayList<>();  // do a minimum age query
  for(Dog d: dogList)
    if(d.getAge() >= testFor)                  // the key moment!
      result1.add(d);
  return result1;
}
static ArrayList<Dog> maxWeight(ArrayList<Dog> dogList, int testFor) {
  ArrayList<Dog> result1 = new ArrayList<>(); // do a max weight query
  for(Dog d: dogList)
    if(d.getWeight() <= testFor)              // the key moment!
      result1.add(d);
  return result1;
} }
```

which produces the following (predictable we hope) output:

```
all dogs [boi, tyri, charis, aiko, clover, mia, zooey]
min age 7 [tyri, charis, aiko, clover, zooey]
max wght. [boi, tyri, clover, mia]
```

You're probably way ahead of us here, but notice how similar the `minAge()` and `maxWeight()` methods are. And it should be easy to imagine other similar methods with names like `maxAge()` or `namesStartingWithC()`. So the line of code we want to focus on is:

```
if( d.getAge() >= testFor ) // the query expression is in bold
```

What if—just sayin'—we could create a single `Dog`-querying method (instead of the many we've been contemplating) and pass it the query expression we wanted it to use? It would sort of look like this:

```
static ArrayList<Dog> dogQuerier(ArrayList<Dog> dogList, // query expression) {
  // do an "on the fly" query
  ArrayList<Dog> result1 = new ArrayList<>();
  for(Dog d: dogList)
    if( // query expression )                          // the key moment!
      result1.add(d);
  return result1;
}
```

Now we're thinking about passing code as an argument? Lambdas let us do just that! Let's look at our `dogQuerier()` method a little more closely. First off, the code we're going to pass in is going to be used as the expression in an `if` statement. What do we know about `if` expressions? Right! They have to resolve to a `boolean` value. So when we declare our method, the second argument (the one that's going to hold the passed-in-code) has to be declared as a `boolean`. The folks who brought us Java 8 and lambdas and streams provided a bunch of new interfaces in the API, and one of the most useful of these is the `java.util.function.Predicate` interface. The `Predicate` interface has some of those new-fangled `static` and `default` interface methods, we discussed earlier in the book, and, most importantly for us, it has one nonconcrete method called `test()` that returns—you guessed it—a `boolean`.

Here's the multipurpose `dogQuerier()` method:

```
static ArrayList<Dog> dogQuerier(ArrayList<Dog> dogList, Predicate<Dog> expr) {
    // do an "on the fly" query
    ArrayList<Dog> result1 = new ArrayList<>();
    for(Dog d: dogList)
      if( expr.test(d) )                          // the key moment!
        result1.add(d);
    return result1;
}
```

So far this looks like good-old Java; it's when we invoke `dogQuerier()` that the syntax gets interesting:

```
dogQuerier(dogs, d -> d.getAge() < 9);
```

When we say [c]d **`-> d.getAge() < 9`**—THAT is the lambda expression. The d represents the argument, and then the code must return a `boolean`. Let's put all of this together in a new version of `TestDogs`:

```
import java.util.*;
import java.util.function.Predicate;
public class TestDogs {
  public static void main(String[] args) {
    ArrayList<Dog> dogs = new ArrayList<>(); // create and populate
    dogs.add(new Dog("boi", 30, 6)); dogs.add(new Dog("tyri", 40, 12));
    dogs.add(new Dog("charis", 120, 7)); dogs.add(new Dog("aiko", 50, 10));
    dogs.add(new Dog("clover", 35, 12)); dogs.add(new Dog("mia", 15, 4));
    dogs.add(new Dog("zooey", 45, 8));
                                         // run a few old "queries"
    System.out.println("all dogs " + dogs);
    System.out.println("min age 7 " + minAge(dogs, 7).toString());
    System.out.println("max wght. " + maxWeight(dogs,40).toString());
                                         // run a few lambda queries
    System.out.println("age < 9 " + dogQuery(dogs, d -> d.getAge() < 9));
    System.out.println("w > 100 " + dogQuery(dogs, d -> d.getWeight() > 100));
  }
  // declare old style "query" methods
  static ArrayList<Dog> minAge(ArrayList<Dog> dogList, int testFor) {
    // do a minimum age query
    ArrayList<Dog> result1 = new ArrayList<>();
    for(Dog d: dogList)
```

```
        if(d.getAge() >= testFor)            // the key moment!
          result1.add(d);
      return result1;
    }
    static ArrayList<Dog> maxWeight(ArrayList<Dog> dogList, int testFor) {
// do a max weight query
      ArrayList<Dog> result1 = new ArrayList<>();
      for(Dog d: dogList)
        if(d.getWeight() <= testFor)          // the key moment!
          result1.add(d);
      return result1;
    }

    // declare a new lambda powered, generic, multi-purpose query method
    static ArrayList< >Dog> dogQuery(ArrayList< >Dog> dogList, Predicate< >Dog> expr) {
      // do an "on the fly" query
      ArrayList< >Dog> result1 = new ArrayList< >>();
      for(Dog d: dogList)
        if(expr.test(d))              // the key moment, lambda powered!
          result1.add(d);
      return result1;
    }
}
```

which produces the following output (the last two lines generated using lambdas!):

```
all dogs [boi, tyri, charis, aiko, clover, mia, zooey]
min age 7 [tyri, charis, aiko, clover, zooey]
max wght. [boi, tyri, clover, mia]
age < > 9 [boi, charis, mia, zooey]
w > 100 [charis]
```

Let's step back now and cover some syntax rules. The following rules are for the purposes of the OCA 8 exam only! If you decide to earn your OCP 8, you'll do a much deeper dive into lambdas, and there are lots of "cans of worms" you'll have to open, which we're purposely going to avoid. So what follows is an OCA 8 appropriate simplification.

The basic syntax for a `Predicate` lambda has three parts:

A Single Parameter	An Arrow-Token	A Body
x	→	7 < > 5

Other types of lambdas take zero or more parameters, but for the OCA 8, we're focused exclusively on the `Predicate`, which must take exactly one parameter. Here are some detailed syntax rules for `Predicate` lambdas:

- The parameter can be just a variable name, or it can be the type followed by a variable name all in parentheses.
- The body MUST (one way or another) return a `boolean`.
- The body can be a single expression, which cannot have a `return` statement.
- The body can be a code block surrounded by curly braces, containing one or more valid statements, each ending with a semicolon, and the block must end with a `return` statement.

Following is a code listing that shows examples of legal and then illegal examples of `Predicate` lambdas:

```
import java.util.function.Predicate;          // type of lambda
                                              // we're learning
public class Lamb2 {
  public static void main(String[] args) {
    Lamb2 m1 = new Lamb2();
// ==== LEGAL LAMBDAS ========================

    m1.go(x -> 7 < 5);                                  // extra terse
    m1.go(x -> { return adder(2, 1) > 5; });            // block
    m1.go((Lamb2 x) -> { int y = 5;
                         return adder(y, 7) > 8; });    // multi-stmt block
    m1.go(x -> { int y=5; return adder(y,6) > 8; });    // no arg type, block
    int a = 5; int b = 6;
    m1.go(x -> { return adder(a, b) > 8; });            // in scope vars
    m1.go((Lamb2 x) -> adder(a, b) > 13);               // arg type, no block
// ==== ILLEGAL LAMBDAS =======================
    // m1.go(x -> return adder(2, 1) > 5; ); // return w/o block
    // m1.go(Lamb2 x -> adder(2, 3) > 7); // type needs parens
    // m1.go(() -> adder(2, 3) > 7); // Predicate needs 1 arg
    // m1.go(x -> { adder(4, 2) > 9 }); // blocks need statements
    // m1.go(x -> { int y = 5; adder(y, 7) > 8; }); // block needs return
 }
 void go(Predicate<Lamb2> e) { // go() takes a predicate
   Lamb2 m2 = new Lamb2();
   System.out.println(e.test(m2) ? "ternary true" // ternary uses boolean expr
                                 : "ternary false");
 }
 static int adder(int x, int y) { return x + y; } // complex calculation
}
```

This code is mostly about valid and invalid syntax, but let's look a little more closely at the `go()` method. The test is mainly concerned with the code to be passed to a method, but it's useful to look (but not TOO closely) at a method that receives lambda code. In both the Dogs code and the code directly above, the receiving method took a `Predicate`. Inside the receiving methods, we created an object of the type we're working with, which we pass to the `Predicate.test()` method. The receiving method expects the `test()` method to return a `boolean`.

We have to admit that lambdas are a bit tricky to learn. Again, we expect we've left you with some unanswered questions, but we think Oracle did a reasonable job of slicing out a piece of the lambda puzzle to start with. If you understand the bits we've covered, you should be able to handle the lambda-related questions Oracle throws you.

CERTIFICATION SUMMARY

The most important thing to remember about `Strings` is that `String` objects are immutable, but references to `Strings` are not! You can make a new `String` by using an existing `String` as a starting point, but if you don't assign a reference variable to the new `String`, it will be lost to your program—you will have no way to access your new `String`. Review the important methods in the `String` class.

The `StringBuilder` class was added in Java 5. It has exactly the same methods as the old `StringBuffer` class, except `StringBuilder`'s methods aren't thread-safe. Because `StringBuilder`'s methods are not thread-safe, they tend to run faster than `StringBuffer` methods, so choose `StringBuilder` whenever threading is not an issue. Both `StringBuffer` and `StringBuilder` objects can have their value changed over and over without your having to create new objects. If you're doing a lot of string manipulation, these objects will be more efficient than immutable `String` objects, which are, more or less, "use once, remain in memory forever." Remember, these methods ALWAYS change the invoking object's value, even with no explicit assignment.

Next we discussed key classes and interfaces in the new Java 8 calendar and time-related packages. Similar to `Strings`, all of the calendar classes we studied create immutable objects. In addition, these classes use factory methods exclusively to create new objects. The keyword `new` cannot be used with these classes. We looked at some of the powerful features of these classes, like calculating the amount of time between two different dates or times. Then we took a look at how the `DateTimeFormatter` class is used to parse `Strings` into calendar objects and how it is used to beautify calendar objects.

The next topic was arrays. We talked about declaring, constructing, and initializing one-dimensional and multidimensional arrays. We talked about anonymous arrays and the fact that arrays of objects are actually arrays of references to objects.

Next, we discussed the basics of `ArrayLists`. `ArrayLists` are like arrays with superpowers that allow them to grow and shrink dynamically and to make it easy for you to insert and delete elements at locations of your choosing within the list. We discussed the idea that `ArrayLists` cannot hold primitives, and that if you want to make an `ArrayList` filled with a given type of primitive values, you use "wrapper" classes to turn a primitive value into an object that represents that value. Then we discussed how with autoboxing, turning primitives into wrapper objects, and vice versa, is done automatically.

Finally, we discussed a specific subset of the topic of lambdas, using the `Predicate` interface. The basic idea of lambdas is that you can pass a bit of code from one method to another. The `Predicate` interface is one of many "functional interfaces" provided in the Java 8 API. A functional interface is one that has only one method to be implemented. In the case of the `Predicate` interface, this method is called `test()`, and it takes a single argument and returns a `boolean`. To wrap up our discussion of lambdas, we covered some of the tricky syntax rules you need to know to write valid lambdas.

TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Using String and StringBuilder (OCA Objectives 9.2 and 9.1)

- `String` objects are immutable, and `String` reference variables are not.
- If you create a new `String` without assigning it, it will be lost to your program.
- If you redirect a `String` reference to a new `String`, the old `String` can be lost.
- `String` methods use zero-based indexes, except for the second argument of `substring()`.

- The `String` class is `final`—it cannot be extended.
- When the JVM finds a `String` literal, it is added to the `String` literal pool.
- Strings have a *method* called `length()`—arrays have an *attribute* named `length`.
- `StringBuilder` objects are mutable—they can change without creating a new object.
- `StringBuilder` methods act on the invoking object, and objects can change without an explicit assignment in the statement.
- Remember that chained methods are evaluated from left to right.
- `String` methods to remember: `charAt()`, `concat()`, `equalsIgnoreCase()`, `length()`, `replace()`, `substring()`, `toLowerCase()`, `toString()`, `toUpperCase()`, and `trim()`.
- `StringBuilder` methods to remember: `append()`, `delete()`, `insert()`, `reverse()`, and `toString()`.

Manipulating Calendar Data (OCA Objective 9.3)

- On the exam all the objects created using the calendar classes are immutable, but their reference variables are not.
- If you create a new calendar object without assigning it, it will be lost to your program.
- If you redirect a calendar reference to a new calendar object, the old calendar object can be lost.
- All of the objects created using the exam's calendar classes must be created using factory methods (e.g., `from()`, `now()`, `of()`, `parse()`); the keyword `new` is not allowed.
- The `until() and between()` methods perform complex calculations that determine the amount of time between the values of two calendar objects.
- The `DateTimeFormatter` class uses the `parse()` method to parse input Strings into valid calendar objects.
- The `DateTimeFormatter` class uses the `format()` method to format calendar objects into beautifully formed Strings.

Using Arrays (OCA Objectives 4.1 and 4.2)

- Arrays can hold primitives or objects, but the array itself is always an object.
- When you declare an array, the brackets can be to the left or right of the 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` attribute 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—for example, 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`).

Using ArrayList (OCA Objective 9.4)

- `ArrayLists` allow you to resize your list and make insertions and deletions to your list far more easily than arrays.
- `ArrayLists` are ordered by default. When you use the `add()` method with no index argument, the new entry will be appended to the end of the `ArrayList`.
- For the OCA 8 exam, the only `ArrayList` declarations you need to know are of this form:

```
ArrayList<type> myList = new ArrayList<type>();
List<type> myList2 = new ArrayList<type>(); // polymorphic
List<type> myList3 = new ArrayList<>(); // diamond operator, polymorphic
optional
```

- `ArrayLists` can hold only objects, not primitives, but remember that autoboxing can make it look like you're adding primitives to an `ArrayList` when, in fact, you're adding a wrapper object version of a primitive.
- An `ArrayList`'s index starts at 0.
- `ArrayLists` can have duplicate entries. Note: Determining whether two objects are duplicates is trickier than it seems and doesn't come up until the OCP 8 exam.
- `ArrayList` methods to remember: `add(element)`, `add(index, element)`, `clear()`, `contains(object)`, `get(index)`, `indexOf(object)`, `remove(index)`, `remove(object)`, and `size()`.

Encapsulating Reference Variables (OCA Objective 6.5)

- If you want to encapsulate mutable objects like `StringBuilders` or arrays or `ArrayLists`, you cannot return a reference to these objects; you must first make a copy of the object and return a reference to the copy.
- Any class that has a method that returns a reference to a mutable object is breaking encapsulation.

Using Predicate Lambda Expressions (OCA Objective 9.5)

- Lambdas allow you to pass bits of code from one method to another. And the receiving method can run whatever complying code it is sent.
- While there are many types of lambdas that Java 8 supports, for this exam, the only lambda type you need to know is the `Predicate`.
- The `Predicate` interface has a single method to implement that's called `test()`, and it takes one argument and returns a `boolean`.
- As the `Predicate.test()` method returns a boolean, it can be placed (mostly?) wherever a boolean expression can go, e.g., in `if`, `while`, `do`, and ternary statements.
- `Predicate` lambda expressions have three parts: a single argument, an arrow (`->`), and an expression or code block.
- A `Predicate` lambda expression's argument can be just a variable or a type and variable together in parentheses, e.g., `(MyClass m)`.
- A `Predicate` lambda expression's body can be an expression that resolves to a boolean, OR it can be a block of statements (surrounded by curly braces) that ends with a boolean-returning `return` statement.

SELF TEST

1. Given: ?

```
public class Mutant {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("abc");
    String s = "abc";
    sb.reverse().append("d");
    s.toUpperCase().concat("d");
    System.out.println("." + sb + "··" + s + ".");
  }
}
```

Which two substrings will be included in the result? (Choose two.)

a. `.abc.`

b. `.ABCd.`

c. `.ABCD.`

d. `.cbad.`

e. `.dcba.`

2. Given: ?

```
public class Hilltop {
  public static void main(String[] args) {
    String[] horses = new String[5];
    horses[4] = null;
    for(int i = 0; i < horses.length; i++) {
      if(i < args.length)
        horses[i] = args[i];
      System.out.print(horses[i].toUpperCase() + " ");
    }
  }
}
```

And, if the code compiles, the command line:

```
java Hilltop eyra vafi draumur kara
```

What is the result?

a. `EYRA VAFI DRAUMUR KARA`

b. `EYRA VAFI DRAUMUR KARA null`

c. An exception is thrown with no other output

d. `EYRA VAFI DRAUMUR KARA`, and then a `NullPointerException`

e. `EYRA VAFI DRAUMUR KARA`, and then an `ArrayIndexOutOfBoundsException`

f. Compilation fails

3. Given: ?

```
public class Actors {
  public static void main(String[] args) {
    char[] ca = {0x4e, \u004e, 78};
    System.out.println((ca[0] == ca[1]) + " " + (ca[0] == ca[2]));
  }
}
```

What is the result?

a. `true true`

b. `true false`

c. `false true`

d. `false false`

e. Compilation fails

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? (Choose all that apply.)

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:

```
import java.util.*;
public class Sequence {
  public static void main(String[] args) {
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("apple");
    myList.add("carrot");
    myList.add("banana");
    myList.add(1, "plum");
    System.out.print(myList);
  }
}
```

What is the result?

a. `[apple, banana, carrot, plum]`

b. `[apple, plum, carrot, banana]`

c. `[apple, plum, banana, carrot]`

d. `[plum, banana, carrot, apple]`

e. `[plum, apple, carrot, banana]`

f. `[banana, plum, carrot, apple]`

g. Compilation fails

6. Given:

```
 3. class Dozens {
 4. int[] dz = {1,2,3,4,5,6,7,8,9,10,11,12};
 5. }
 6. public class Eggs {
 7.    public static void main(String[] args) {
 8.      Dozens [] da = new Dozens[3];
 9.      da[0] = new Dozens();
10.      Dozens d = new Dozens();
11.      da[1] = d;
12.      d = null;
13.      da[1] = null;
14.      // do stuff
15.    }
16. }
```

Which two are true about the objects created within `main()`, and which are eligible for garbage collection when line 14 is reached?

a. Three objects were created

b. Four objects were created

c. Five objects were created

d. Zero objects are eligible for GC

e. One object is eligible for GC

f. Two objects are eligible for GC

g. Three objects are eligible for GC

7. Given:

```
public class Tailor {
  public static void main(String[] args) {
    byte[][] ba = {{1,2,3,4}, {1,2,3}};
    System.out.println(ba[1].length + " " + ba.length);
  }
}
```

What is the result?

a. `2 4`

b. `2 7`

c. `3 2`

d. `3 7`

e. `4 2`

f. `4 7`

g. Compilation fails

8. Given: ?

```
public class Theory {
  public static void main(String[] args) {
    String s1 = "abc";
    String s2 = s1;
    s1 += "d";
    System.out.println(s1 + " " + s2 + " " + (s1==s2));

    StringBuilder sb1 = new StringBuilder("abc");
    StringBuilder sb2 = sb1;
    sb1.append("d");
    System.out.println(sb1 + " " + sb2 + " " + (sb1==sb2));
  }
}
```

Which are true? (Choose all that apply.)

a. Compilation fails

b. The first line of output is `abc abc true`

c. The first line of output is `abc abc false`

d. The first line of output is `abcd abc false`

e. The second line of output is `abcd abc false`

f. The second line of output is `abcd abcd true`

g. The second line of output is `abcd abcd false`

9. Given: ?

```
public class Mounds {
 public static void main(String[] args) {
   StringBuilder sb = new StringBuilder();
   String s = new String();
   for(int i = 0; i < 1000; i++) {
     s = " " + i;
     sb.append(s);
   }
   // done with loop
 }
}
```

If the garbage collector does NOT run while this code is executing, approximately how many objects will exist in memory when the loop is done?

a. Less than 10

b. About 1000

c. About 2000

d. About 3000

e. About 4000

10. Given: ?

```
class Box {
  int size;
  Box(int s) { size = s; }
}
```

```
 7. public class Laser {
 8.   public static void main(String[] args) {
 9.     Box b1 = new Box(5);
10.     Box[] ba = go(b1, new Box(6));
11.     ba[0] = b1;
12.     for(Box b : ba) System.out.print(b.size + " ");
13.   }
14.   static Box[] go(Box b1, Box b2) {
15.      b1.size = 4;
16.      Box[] ma = {b2, b1};
17.      return ma;
18.   }
19. }
```

What is the result?

a. `4 4`

b. `5 4`

c. `6 4`

d. `4 5`

e. `5 5`

f. Compilation fails

11. Given: ?

```
public class Hedges {
  public static void main(String[] args) {
    String s = "JAVA";
    s = s + "rocks";
    s = s.substring(4,8);
    s.toUpperCase();
    System.out.println(s);
  }
}
```

What is the result?

a. `JAVA`

b. `JAVAROCKS`

c. `rocks`

d. `rock`

e. `ROCKS`

f. `ROCK`

g. Compilation fails

12. Given: ?

```
 1. import java.util.*;
 2. class Fortress {
 3.   private String name;
 4.   private ArrayList<Integer> list;
 5.   Fortress() { list = new ArrayList<Integer>(); }
 6.
 7.   String getName() { return name; }
 8.   void addToList(int x) { list.add(x); }
 9.   ArrayList getList() { return list; }
10. }
```

Which lines of code (if any) break encapsulation? (Choose all that apply.)

a. Line 3

b. Line 4

c. Line 5

d. Line 7

e. Line 8

f. Line 9

g. The class is already well encapsulated

13. Given: ?

```
 import java.util.function.Predicate;
public class Sheep {
  public static void main(String[] args) {
    Sheep s = new Sheep();
    s.go(() -> adder(5, 1) < 7); // line A
    s.go(x -> adder(6, 2) < 9); // line B
    s.go(x, y -> adder(3, 2) < 4); // line C
  }
  void go(Predicate<Sheep> e) {
    Sheep s2 = new Sheep();
    if(e.test(s2))
      System.out.print("true ");
    else
      System.out.print("false ");
  }
  static int adder(int x, int y) {
    return x + y;
  }
}
```

What is the result?

a. `true true false`

b. Compilation fails due only to an error at line A

c. Compilation fails due only to an error at line B

d. Compilation fails due only to an error at line C

e. Compilation fails due only to errors at lines A and B

f. Compilation fails due only to errors at lines A and C

g. Compilation fails due only to errors at lines A, B, and C

h. Compilation fails for reasons not listed

14. Given: ?

```
import java.time.*;
import java.time.format.*;
public class Shiny {
  public static void main(String[] args) {
    DateTimeFormatter f1 =
              DateTimeFormatter.ofPattern("MMM dd, yyyy");
    LocalDate d = LocalDate.of(2018, Month.JANUARY, 15);
    LocalDate d2 = d.plusDays(1);
    System.out.print(f1.format(d) + " ");
    System.out.println(d2.format(f1));
  }
}
```

What is the result?

a. `2018-01-15 2018-01-15`

b. `2018-01-15 2018-01-16`

c. `Jan 15, 2018 Jan 15, 2018`

d. `Jan 15, 2018 Jan 16, 2018`

e. Compilation fails

f. An exception is thrown at runtime

15. Given: ?

```
import java.util.*;
public class Jackets {
  public static void main(String[] args) {
    List<Integer> myList = new ArrayList<>();   // line 5
    myList.add(new Integer(5));
    myList.add(42);                             // line 7
    myList.add("113");                          // line 8
    myList.add(new Integer("7"));               // line 9
    System.out.println(myList);
  }
}
```

What is the result?

a. `[5, 42, 113, 7]`

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

c. Compilation fails due only to an error on line 8

d. Compilation fails due only to errors on lines 5 and 8

e. Compilation fails due only to errors on lines 7 and 8

f. Compilation fails due only to errors on lines 5, 7, and 8

g. Compilation fails due only to errors on lines 5, 7, 8, and 9

16. Given that `adder()` returns an `int`, which are valid `Predicate` lambdas? (Choose all that apply.) ?

a. `x, y -> 7 < 5`

b. `x -> { return adder(2, 1) > 5; }`

c. `x -> return adder(2, 1) > 5;`

d.
```
x -> { int y = 5;
       int z = 7;
       adder(y, z) > 8; }
```

e.
```
x -> { int y = 5;
       int z = 7;
       return adder(y, z) > 8; }
```

f. `(MyClass x) -> 7 > 13`

g. `(MyClass x) -> 5 + 4`

17. Given: ?

```
import java.util.*;
public class Baking {
  public static void main(String[] args) {
    ArrayList<String> steps = new ArrayList<String>();
    steps.add("knead");
    steps.add("oil pan");
    steps.add("turn on oven");
    steps.add("roll");
    steps.add("turn on oven");
    steps.add("bake");
    System.out.println(steps);
  }
}
```

What is the result?

a. `[knead, oil pan, roll, turn on oven, bake]`

b. `[knead, oil pan, turn on oven, roll, bake]`

c. `[knead, oil pan, turn on oven, roll, turn on oven, bake]`

d. The output is unpredictable

e. Compilation fails

f. An exception is thrown at runtime

18. Given: ?

```
import java.time.*;
public class Bachelor {
  public static void main(String[] args) {
    LocalDate d = LocalDate.of(2018, 8, 15);
    d = d.plusDays(1);
    LocalDate d2 = d.plusDays(1);
    LocalDate d3 = d2;
    d2 = d2.plusDays(1);
    System.out.println(d + " " + d2 + " " + d3); // line X
  }
}
```

Which are true? (Choose all that apply.)

a. The output is: `2018-08-16 2018-08-17 2018-08-18`

b. The output is: `2018-08-16 2018-08-18 2018-08-17`

c. The output is: `2018-08-16 2018-08-17 2018-08-17`

d. At line X, zero `LocalDate` objects are eligible for garbage collection

e. At line X, one `LocalDate` object is eligible for garbage collection

f. At line X, two `LocalDate` objects are eligible for garbage collection

g. Compilation fails

19. **19**. Given that e refers to an object that implements `Predicate`, which could be valid code snippets or statements? (Choose all that apply.) ?

a. `if(e.test(m))`

b. `switch (e.test(m))`

c. `while(e.test(m))`

d. `e.test(m) ? "yes" : "no";`

e. `do {} while(e.test(m));`

f. `System.out.print(e.test(m));`

g. `boolean b = e.test(m));`

Answers

1. ☑ **A** and **D** are correct. The `String` operations are working on a new (lost) `String` not `String s`. The `StringBuilder` operations work from left to right.

☒ **B**, **C**, and **E** are incorrect based on the above. (OCA Objectives 9.2 and 9.1)

2. ☑ **D** is correct. The `horses` array's first four elements contain `Strings`, but the fifth is null, so the `toUpperCase()` invocation for the fifth element throws a `NullPointerException`.

☒ **A, B, C, E**, and **F** are incorrect based on the above. (OCA Objectives 4.1 and 1.3)

3. ☑ **E** is correct. The Unicode declaration must be enclosed in single quotes: `'\u004e'`. If this were done, the answer would be **A**, but that equality isn't on the OCA exam.

☒ **A, B, C**, and **D** are incorrect based on the above. (OCA Objectives 2.1 and 4.1)

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

☒ **A, B, D, E, F**, and **G** are incorrect based on the above. (OCA Objective 4.2)

5. ☑ **B** is correct. `ArrayList` elements are automatically inserted in the order of entry; they are not automatically sorted. `ArrayLists` use zero-based indexes, and the last `add()` inserts a new element and shifts the remaining elements back.

   ☒ **A, C, D, E, F**, and **G** are incorrect based on the above. (OCA Objective 9.4)

6. ☑ **C** and **F** are correct. `da` refers to an object of type "`Dozens` array," and each `Dozens` object that is created comes with its own "`int` array" object. When line 14 is reached, only the second `Dozens` object (and its "`int` array" object) are not reachable.

   ☒ **A, B, D, E**, and **G** are incorrect based on the above. (OCA Objectives 4.1 and 2.4)

7. ☑ **C** is correct. A two-dimensional array is an "array of arrays." The length of `ba` is 2 because it contains 2 one-dimensional arrays. Array indexes are zero-based, so `ba[1]` refers to `ba`'s second array.

   ☒ **A, B, D, E, F**, and **G** are incorrect based on the above. (OCA Objective 4.2)

8. ☑ **D** and **F** are correct. Although `String` objects are immutable, references to `Strings` are mutable. The code `s1 += "d";` creates a new `String` object. `StringBuilder` objects are mutable, so the `append()` is changing the single `StringBuilder` object to which both `StringBuilder` references refer.

   ☒ **A, B, C, E**, and **G** are incorrect based on the above. (OCA Objectives 9.2 and 9.1)

9. ☑ **B** is correct. `StringBuilders` are mutable, so all of the `append()` invocations are acting on the same `StringBuilder` object over and over. `Strings`, however, are immutable, so every `String` concatenation operation results in a new `String` object. Also, the string `" "` is created once and reused in every loop iteration.

   ☒ **A, C, D**, and **E** are incorrect based on the above. (OCA Objectives 9.2 and 9.1)

10. ☑ **A** is correct. Although `main()`'s `b1` is a different reference variable than `go()`'s `b1`, they refer to the same `Box` object.

    ☒ **B, C, D, E**, and **F** are incorrect based on the above. (OCA Objectives 4.1, 6.1, and 6.6)

11. ☑ **D** is correct. The `substring()` invocation uses a zero-based index and the second argument is exclusive, so the character at index 8 is NOT included. The `toUpperCase()` invocation makes a new `String` object that is instantly lost. The `toUpperCase()` invocation does NOT affect the `String` referred to by `s`.

    ☒ **A, B, C, E, F**, and **G** are incorrect based on the above. (OCA Objective 9.2)

12. ☑ **F** is correct. When encapsulating a mutable object like an `ArrayList`, your getter must return a reference to a copy of the object, not just the reference to the original object.

    ☒ **A, B, C, D, E**, and **G** are incorrect based on the above. (OCA Objective 6.5)

13. ☑ **F** is correct. Predicate lambdas take exactly one parameter; the rest of the code is correct.

    ☒ **A, B, C, D, E, G**, and **H** are incorrect based on the above. (OCA Objective 9.5)

14. ☑ **D** is correct. Invoking the `plusDays()` method creates a new object, and both `LocalDate` and `DateTimeFormatter` have `format()` methods.

    ☒ **A, B, C, E**, and **F** are incorrect based on the above. (OCA Objective 9.3)

15. ☑ **C** is correct. The only error in this code is attempting to add a `String` to an `ArrayList` of `Integer` wrapper objects. Line 7 uses autoboxing, and lines 6 and 9 demonstrate using a wrapper class's two constructors.

    ☒ **A, B, D, E, F**, and **G** are incorrect based on the above. (OCA Objectives 2.5 and 9.4)

16. ☑ **B, E** and **F** use correct syntax.

☒ **A, C, D**, and **G** are incorrect. **A** passes two parameters. **C**, a `return`, must be in a code block, and code blocks must be in curly braces. **D**, a block, must have a `return` statement. **G**, the result, is not a `boolean`. (OCA Objective 9.5)

17. ☑ **C** is correct. `ArrayLists` can have duplicate entries.

☒ **A, B, D, E**, and **F** are incorrect based on the above. (OCA Objective 9.4)

18. ☑ **B** and **E** are correct. A total of four `LocalDate` objects are created, but the one created using the `of()` method is abandoned on the next line of code when its reference variable is assigned to the new `LocalDate` object created via the first `plusDays()` invocation. The reference variables are swapped a bit, which accounts for the dates not printing in chronological order.

☒ **A, C, D, F**, and **G** are incorrect based on the above. (OCA Objectives 2.4 and 9.3)

19. ☑ **A, C, D, E, F**, and **G** are correct; they all require a `boolean`.

☒ **B** is incorrect. A `switch` doesn't take a `boolean`. (OCA Objective 9.5)