Chapter 5
Core Java APIs

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Working with Java Primitive Data Types and String APIs**
  - Create and manipulate Strings
  - Manipulate data using the StringBuilder class and its methods
- **Working with Java Arrays**
  - Declare, instantiate, initialize and use a one-dimensional array
  - Declare, instantiate, initialize and use a two-dimensional array
- **Programming Abstractly Through Interfaces**
  - Declare and use List and ArrayList instances

In the context of an Application Programming Interface (API), an interface refers to a group of classes or Java interface definitions giving you access to a service or functionality.

In this chapter, you will learn about many core data structures in Java, along with the most common APIs to access them. For example, `String` and `StringBuilder`, along with their associated APIs, are used to create and manipulate text data. An array, `List`, `Set`, or `Map` are used to manage often large groups of data. You'll also learn how to determine whether two objects are equivalent.

This chapter is long, so we recommend reading it in multiple sittings. On the bright side, it contains most of the APIs you need to know for the exam.

Creating and Manipulating Strings

The `String` class is such a fundamental class that you'd be hard-pressed to write code without it. After all, you can't even write a `main()` method without using the `String` class. A *string* is basically a sequence of characters; here's an example:

```
String name = "Fluffy";
```

As you learned in Chapter 2, "Java Building Blocks," this is an example of a reference type. You also learned that reference types are created using the `new` keyword. Wait a minute. Something is missing from the previous example: It doesn't have `new` in it! In Java, these two snippets both create a `String`:

```
String name = "Fluffy";
String name = new String("Fluffy");
```

Both give you a reference variable named `name` pointing to the `String` object `"Fluffy"`. They are subtly different, as you'll see in the section "The String Pool" later in this chapter. For now, just remember that the `String` class is special and doesn't need to be instantiated with `new`.

Since a `String` is a sequence of characters, you probably won't be surprised to hear that it implements the interface `CharSequence`. This interface is a general way of representing several classes, including `String` and `StringBuilder`. You'll learn more about interfaces later in the book.

In this section, we'll look at concatenation, immutability, common methods, and method chaining.

Concatenation

In Chapter 3, "Operators," you learned how to add numbers. 1 + 2 is clearly 3. But what is `"1" + "2"`? It's actually `"12"` because Java combines the two `String` objects. Placing one `String` before the other `String` and combining them is called string *concatenation*. The exam creators like string concatenation because the `+` operator can be used in two ways within the same line of code. There aren't a lot of rules to know for this, but you have to know them well:

1. If both operands are numeric, `+` means numeric addition.
2. If either operand is a `String`, `+` means concatenation.
3. The expression is evaluated left to right.

Now let's look at some examples:

```
System.out.println(1 + 2);           // 3
System.out.println("a" + "b");       // ab
System.out.println("a" + "b" + 3);   // ab3
System.out.println(1 + 2 + "c");     // 3c
System.out.println("c" + 1 + 2);     // c12
```

The first example uses the first rule. Both operands are numbers, so we use normal addition. The second example is simple string concatenation, described in the second rule. The quotes for the `String` are only used in code—they don't get output.

The third example combines both the second and third rules. Since we start on the left, Java figures out what `"a" + "b"` evaluates to. You already know that one: It's `"ab"`. Then Java looks at the remaining expression of `"ab" + 3`. The second rule tells us to concatenate since one of the operands is a `String`.

In the fourth example, we start with the third rule, which tells us to consider `1 + 2`. Both operands are numeric, so the first rule tells us the answer is `3`. Then we have `3 + "c"`, which uses the second rule to give us `"3c"`. Notice all three rules get used in one line?

Finally, the fifth example shows the importance of the third rule. First we have `"c" + 1`, which uses the second rule to give us `"c1"`. Then we have `"c1" + 2`, which uses the second rule again to give us `"c12"`.

The exam takes this a step further and will try to trick you with something like this:

```
int three = 3;
String four = "4";
System.out.println(1 + 2 + three + four);
```

When you see this, just take it slow and remember the three rules—and be sure to check the variable types. In this example, we start with the third rule, which tells us to consider `1 + 2`. The first rule gives us `3`. Next we have `3 + three`. Since `three` is of type `int`, we still use the first rule, giving us `6`. Next we have `6 + four`. Since `four` is of type `String`, we switch to the second rule and get a final answer of `"64"`. When you see questions like this, just take your time and check the types. Being methodical pays off.

There is only one more thing to know about concatenation, but it is an easy one. In this example, you just have to remember what `+=` does. `s += "2"` means the same thing as `s = s + "2"`.

```
4: String s = "1";            // s currently holds "1"
5: s += "2";                  // s currently holds "12"
6: s += 3;                    // s currently holds "123"
7: System.out.println(s);     // 123
```

On line 5, we are "adding" two strings, which means we concatenate them. Line 6 tries to trick you by adding a number, but it's just like we wrote `s = s + 3`. We know that a string "plus" anything else means to use concatenation.

To review the rules one more time: Use numeric addition if two numbers are involved, use concatenation otherwise, and evaluate from left to right. Have you memorized these three rules yet? Be sure to do so before the exam!

Immutability

Once a `String` object is created, it is not allowed to change. It cannot be made larger or smaller, and you cannot change one of the characters inside it.

You can think of a `String` as a storage box you have perfectly full and whose sides can't bulge. There's no way to add objects, nor can you replace objects without disturbing the entire arrangement. The trade-off for the optimal packing is zero flexibility.

Mutable is another word for changeable. *Immutable* is the opposite—an object that can't be changed once it's created. On the exam, you need to know that `String` is immutable.

MORE ON IMMUTABILITY

You won't be asked to identify whether custom classes are immutable on the OCP part 1 exam, but it's helpful to see an example. Consider the following code:

```
class Mutable {
   private String s;
   public void setS(String newS){ s = newS; }  // Setter makes it mutable
   public String getS() { return s; }
}
final class Immutable {
   private String s = "name";
   public String getS() { return s; }
}
```

`Immutable` has only a getter. There's no way to change the value of `s` once it's set. `Mutable` has a setter. This allows the reference `s` to change to point to a different `String` later. Note that even though the `String` class is immutable, it can still be used in a mutable class. You can even make the instance variable `final` so the compiler reminds you if you accidentally change `s`.

Also, immutable classes in Java are final, which prevents subclasses creation. You wouldn't want a subclass adding mutable behavior.

You learned that + is used to do `String` concatenation in Java. There's another way, which isn't used much on real projects but is great for tricking people on the exam. What does this print out?

```
String s1 = "1";
String s2 = s1.concat("2");
s2.concat("3");
System.out.println(s2);
```

Did you say `"12"` ? Good. The trick is to see if you forget that the `String` class is immutable by throwing a method call at you.

Important *String* Methods

The `String` class has dozens of methods. Luckily, you need to know only a handful for the exam. The exam creators pick most of the methods developers use in the real world.

For all these methods, you need to remember that a string is a sequence of characters and Java counts from 0 when indexed. Figure 5.1 shows how each character in the string `"animals"` is indexed.

a	n	i	m	a	l	s
0	1	2	3	4	5	6

FIGURE 5.1 Indexing for a string

Let's look at a number of methods from the `String` class. Many of them are straightforward, so we won't discuss them at length. You need to know how to use these methods. We left out `public` from the signatures in the following sections so you can focus on the important parts.

length()

The method `length()` returns the number of characters in the `String` . The method signature is as follows:

```
int length()
```

The following code shows how to use `length()` :

```
String string = "animals";
System.out.println(string.length());  // 7
```

Wait. It outputs 7? Didn't we just tell you that Java counts from 0? The difference is that zero counting happens only when you're using indexes or positions within a list. When determining the total size or length, Java uses normal counting again.

charAt()

The method `charAt()` lets you query the string to find out what character is at a specific index. The method signature is as follows:

```
char charAt(int index)
```

The following code shows how to use `charAt()`:

```
String string = "animals";
System.out.println(string.charAt(0));  // a
System.out.println(string.charAt(6));  // s
System.out.println(string.charAt(7));  // throws exception
```

Since indexes start counting with 0, `charAt(0)` returns the "first" character in the sequence. Similarly, `charAt(6)` returns the "seventh" character in the sequence. `charAt(7)` is a problem. It asks for the "eighth" character in the sequence, but there are only seven characters present. When something goes wrong that Java doesn't know how to deal with, it throws an exception, as shown here. You'll learn more about exceptions in Chapter 10, "Exceptions."

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 7
```

indexOf()

The method `indexOf()` looks at the characters in the string and finds the first index that matches the desired value. `indexOf` can work with an individual character or a whole `String` as input. It can also start from a requested position. Remember that a `char` can be passed to an `int` parameter type. On the exam, you'll only see a `char` passed to the parameters named `ch`. The method signatures are as follows:

```
int indexOf(int ch)

int indexOf(int ch, int fromIndex)
int indexOf(String str)
int indexOf(String str, int fromIndex)
```

The following code shows how to use `indexOf()`:

```
String string = "animals";
System.out.println(string.indexOf('a'));         // 0
System.out.println(string.indexOf("al"));        // 4
System.out.println(string.indexOf('a', 4));      // 4
System.out.println(string.indexOf("al", 5));     // -1
```

Since indexes begin with 0, the first `'a'` matches at that position. The second statement looks for a more specific string, so it matches later. The third statement says Java shouldn't even look at the characters until it gets to index 4. The final statement doesn't find anything because it starts looking after the match occurred. Unlike `charAt()`, the `indexOf()` method doesn't throw an exception if it can't find a match. `indexOf()` returns –1 when no match is found. Because indexes start with 0, the caller knows that –1 couldn't be a valid index. This makes it a common value for a method to signify to the caller that no match is found.

substring()

The method `substring()` also looks for characters in a string. It returns parts of the string. The first parameter is the index to start with for the re-

turned string. As usual, this is a zero-based index. There is an optional second parameter, which is the end index you want to stop at.

Notice we said "stop at" rather than "include." This means the `endIndex` parameter is allowed to be 1 past the end of the sequence if you want to stop at the end of the sequence. That would be redundant, though, since you could omit the second parameter entirely in that case. In your own code, you want to avoid this redundancy. Don't be surprised if the exam uses it, though. The method signatures are as follows:

```
String substring(int beginIndex)
String substring(int beginIndex, int endIndex)
```

It helps to think of indexes a bit differently for the substring methods. Pretend the indexes are right before the character they would point to. Figure 5.2 helps visualize this. Notice how the arrow with the `0` points to the character that would have index `0`. The arrow with the `1` points between characters with indexes `0` and `1`. There are seven characters in the `String`. Since Java uses zero-based indexes, this means the last character has an index of `6`. The arrow with the `7` points immediately after this last character. This will help you remember that `endIndex` doesn't give an out-of-bounds exception when it is one past the end of the `String`.

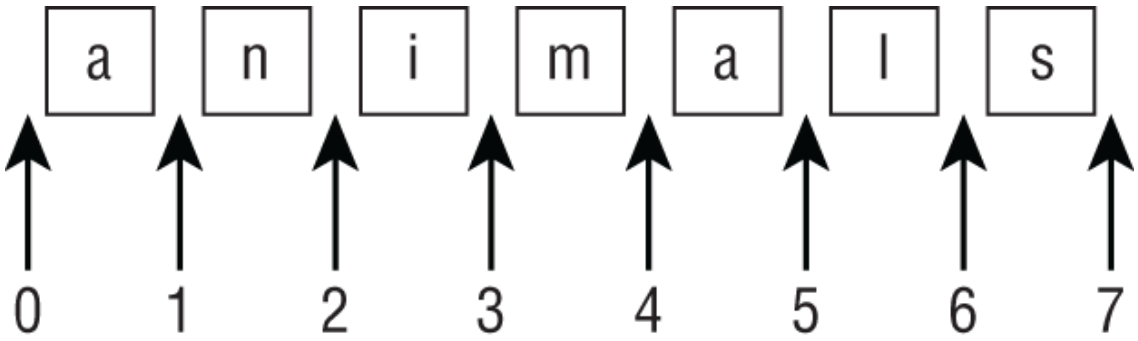

FIGURE 5.2 Indexes for a substring

The following code shows how to use `substring()`:

```
String string = "animals";
System.out.println(string.substring(3));                    // mals
System.out.println(string.substring(string.indexOf('m'))); // mals
System.out.println(string.substring(3, 4));                 // m
System.out.println(string.substring(3, 7));                 // mals
```

The `substring()` method is the trickiest `String` method on the exam. The first example says to take the characters starting with index 3 through the end, which gives us `"mals"`. The second example does the same thing, but it calls `indexOf()` to get the index rather than hard-coding it. This is a common practice when coding because you may not know the index in advance.

The third example says to take the characters starting with index 3 until, but not including, the character at index 4—which is a complicated way of saying we want a `String` with one character: the one at index 3. This results in `"m"`. The final example says to take the characters starting with index 3 until we get to index 7. Since index 7 is the same as the end of the string, it is equivalent to the first example.

We hope that wasn't too confusing. The next examples are less obvious:

```
System.out.println(string.substring(3, 3)); // empty string
System.out.println(string.substring(3, 2));  // throws exception
System.out.println(string.substring(3, 8)); // throws exception
```

The first example in this set prints an empty string. The request is for the characters starting with index 3 until you get to index 3. Since we start and end with the same index, there are *no* characters in between. The second example in this set throws an exception because the indexes can't be backward. Java knows perfectly well that it will never get to index 2 if it starts with index 3. The third example says to continue until the eighth character. There is no eighth position, so Java throws an exception. Granted, there is no seventh character either, but at least there is the "end of string" invisible position.

Let's review this one more time since `substring()` is so tricky. The method returns the string starting from the requested index. If an end index is requested, it stops right before that index. Otherwise, it goes to the end of the string.

toLowerCase() and *toUpperCase()*

Whew. After that mental exercise, it is nice to have methods that do exactly what they sound like! These methods make it easy to convert your data. The method signatures are as follows:

```
String toLowerCase()
String toUpperCase()
```

The following code shows how to use these methods:

```
String string = "animals";
System.out.println(string.toUpperCase());  // ANIMALS
System.out.println("Abc123".toLowerCase());  // abc123
```

These methods do what they say. `toUpperCase()` converts any lowercase characters to uppercase in the returned string. `toLowerCase()` converts any uppercase characters to lowercase in the returned string. These methods leave alone any characters other than letters. Also, remember that strings are immutable, so the original string stays the same.

equals() and *equalsIgnoreCase()*

The `equals()` method checks whether two `String` objects contain exactly the same characters in the same order. The `equalsIgnoreCase()` method checks whether two `String` objects contain the same characters with the exception that it will convert the characters' case if needed. The method signatures are as follows:

```
boolean equals(Object obj)
boolean equalsIgnoreCase(String str)
```

You might have noticed that `equals()` takes an `Object` rather than a `String`. This is because the method is the same for all objects. If you pass in something that isn't a `String`, it will just return `false`. By contrast, the `equalsIgnoreCase` method only applies to `String` objects so it can take the more specific type as the parameter.

The following code shows how to use these methods:

```
System.out.println("abc".equals("ABC"));  // false
System.out.println("ABC".equals("ABC"));  // true
System.out.println("abc".equalsIgnoreCase("ABC"));  // true
```

This example should be fairly intuitive. In the first example, the values aren't exactly the same. In the second, they are exactly the same. In the third, they differ only by case, but it is okay because we called the method that ignores differences in case.

startsWith() and *endsWith()*

The `startsWith()` and `endsWith()` methods look at whether the provided value matches part of the `String`. The method signatures are as follows:

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)
```

The following code shows how to use these methods:

```
System.out.println("abc".startsWith("a")); // true
System.out.println("abc".startsWith("A")); // false
```

```
System.out.println("abc".endsWith("c")); // true
System.out.println("abc".endsWith("a")); // false
```

Again, nothing surprising here. Java is doing a case-sensitive check on the values provided.

replace()

The `replace()` method does a simple search and replace on the string. There's a version that takes `char` parameters as well as a version that takes `CharSequence` parameters. The method signatures are as follows:

```
String replace(char oldChar, char newChar)
String replace(CharSequence target, CharSequence replacement)
```

The following code shows how to use these methods:

```
System.out.println("abcabc".replace('a', 'A')); // AbcAbc
System.out.println("abcabc".replace("a", "A")); // AbcAbc
```

The first example uses the first method signature, passing in `char` parameters. The second example uses the second method signature, passing in `String` parameters.

contains()

The `contains()` method looks for matches in the `String` . It isn't as particular as `startsWith()` and `endsWith()` —the match can be anywhere in the `String` . The method signature is as follows:

```
boolean contains(CharSequence charSeq)
```

The following code shows how to use these methods:

```
System.out.println("abc".contains("b")); // true
System.out.println("abc".contains("B")); // false
```

Again, we have a case-sensitive search in the `String`. The `contains()` method is a convenience method so you don't have to write `str.indexOf(otherString) != -1`.

trim(), strip(), stripLeading(), and stripTrailing()

You've made it through almost all the `String` methods you need to know. Next up is removing blank space from the beginning and/or end of a `String`. The `strip()` and `trim()` methods remove whitespace from the beginning and end of a `String`. In terms of the exam, whitespace consists of spaces along with the `\t` (tab) and `\n` (newline) characters. Other characters, such as `\r` (carriage return), are also included in what gets trimmed. The `strip()` method is new in Java 11. It does everything that `trim()` does, but it supports Unicode.

You don't need to know about Unicode for the exam. But if you want to test the difference, one of Unicode whitespace characters is as follows:

```
char ch = '\u2000';
```

Additionally, the `stripLeading()` and `stripTrailing()` methods were added in Java 11. The `stripLeading()` method removes whitespace from the beginning of the `String` and leaves it at the end. The `stripTrailing()` method does the opposite. It removes whitespace from the end of the `String` and leaves it at the beginning.

The method signatures are as follows:

```
String strip()
String stripLeading()
String stripTrailing()
String trim()
```

The following code shows how to use these methods:

```
System.out.println("abc".strip());                 // abc
System.out.println("\t   a b c\n".strip());        // a b c

String text = " abc\t ";
System.out.println(text.trim().length());         // 3
System.out.println(text.strip().length());        // 3
System.out.println(text.stripLeading().length()); // 5
System.out.println(text.stripTrailing().length());// 4
```

First, remember that `\t` is a single character. The backslash escapes the `t` to represent a tab. The first example prints the original string because there are no whitespace characters at the beginning or end. The second example gets rid of the leading tab, subsequent spaces, and the trailing newline. It leaves the spaces that are in the middle of the string.

The remaining examples just print the number of characters remaining. You can see that both `trim()` and `strip()` leave the same three characters `"abc"` because they remove both the leading and trailing whitespace. The `stripLeading()` method only removes the one whitespace character at the beginning of the `String`. It leaves the tab and space at the end. The `stripTrailing()` method removes these two characters at the end but leaves the character at the beginning of the `String`.

intern()

The `intern()` method returns the value from the string pool if it is there. Otherwise, it adds the value to the string pool. We will explain about the string pool and give examples for `intern()` later in the chapter. The method signature is as follows:

```
String intern()
```

Method Chaining

It is common to call multiple methods as shown here:

```
String start = "AniMaL   ";
String trimmed = start.trim();                 // "AniMaL"
String lowercase = trimmed.toLowerCase();      // "animal"
String result = lowercase.replace('a', 'A');   // "AnimAl"
System.out.println(result);
```

This is just a series of `String` methods. Each time one is called, the returned value is put in a new variable. There are four `String` values along the way, and `AnimAl` is output.

However, on the exam there is a tendency to cram as much code as possible into a small space. You'll see code using a technique called method chaining. Here's an example:

```
String result = "AniMaL   ".trim().toLowerCase().replace('a', 'A');
System.out.println(result);
```

This code is equivalent to the previous example. It also creates four `String` objects and outputs `AnimAl` . To read code that uses method chaining, start at the left and evaluate the first method. Then call the next method on the returned value of the first method. Keep going until you get to the semicolon.

Remember that `String` is immutable. What do you think the result of this code is?

```
5: String a = "abc";
6: String b = a.toUpperCase();
7: b = b.replace("B", "2").replace('C', '3');
8: System.out.println("a=" + a);
9: System.out.println("b=" + b);
```

On line 5, we set `a` to point to `"abc"` and never pointed `a` to anything else. Since we are dealing with an immutable object, none of the code on lines 6 and 7 changes `a`, and the value remains `"abc"`.

`b` is a little trickier. Line 6 has `b` pointing to `"ABC"`, which is straightforward. On line 7, we have method chaining. First, `"ABC".replace("B", "2")` is called. This returns `"A2C"`. Next, `"A2C".replace('C', '3')` is called. This returns `"A23"`. Finally, `b` changes to point to this returned `String`. When line 9 executes, `b` is `"A23"`.

Using the *StringBuilder* Class

A small program can create a lot of `String` objects very quickly. For example, how many do you think this piece of code creates?

```
10: String alpha = "";
11: for(char current = 'a'; current <= 'z'; current++)
12:    alpha += current;
13: System.out.println(alpha);
```

The empty `String` on line 10 is instantiated, and then line 12 appends an `"a"`. However, because the `String` object is immutable, a new `String` object is assigned to `alpha`, and the `""` object becomes eligible for garbage collection. The next time through the loop, `alpha` is assigned a new `String` object, `"ab"`, and the `"a"` object becomes eligible for

garbage collection. The next iteration assigns `alpha` to `"abc"`, and the `"ab"` object becomes eligible for garbage collection, and so on.

This sequence of events continues, and after 26 iterations through the loop, a total of 27 objects are instantiated, most of which are immediately eligible for garbage collection.

This is very inefficient. Luckily, Java has a solution. The `StringBuilder` class creates a `String` without storing all those interim `String` values. Unlike the `String` class, `StringBuilder` is not immutable.

```
15: StringBuilder alpha = new StringBuilder();
16: for(char current = 'a'; current <= 'z'; current++)
17:    alpha.append(current);
18: System.out.println(alpha);
```

On line 15, a new `StringBuilder` object is instantiated. The call to `append()` on line 17 adds a character to the `StringBuilder` object each time through the `for` loop appending the value of current to the end of `alpha`. This code reuses the same `StringBuilder` without creating an interim `String` each time.

In old code, you might see references to `StringBuffer`. It works the same way except it supports threads, which you'll learn about when preparing for the 1Z0-816 exam. `StringBuffer` is no longer on either exam. It performs slower than `StringBuilder`, so just use `StringBuilder`.

In this section, we'll look at creating a `StringBuilder` and using its common methods.

Mutability and Chaining

We're sure you noticed this from the previous example, but `StringBuilder` is not immutable. In fact, we gave it 27 different values in the example (blank plus adding each letter in the alphabet). The exam

will likely try to trick you with respect to `String` and `StringBuilder` being mutable.

Chaining makes this even more interesting. When we chained `String` method calls, the result was a new `String` with the answer. Chaining `StringBuilder` methods doesn't work this way. Instead, the `StringBuilder` changes its own state and returns a reference to itself. Let's look at an example to make this clearer:

```
4: StringBuilder sb = new StringBuilder("start");
5: sb.append("+middle");                      // sb = "start+middle"
6: StringBuilder same = sb.append("+end");    // "start+middle+end"
```

Line 5 adds text to the end of `sb`. It also returns a reference to `sb`, which is ignored. Line 6 also adds text to the end of `sb` and returns a reference to `sb`. This time the reference is stored in `same`—which means `sb` and `same` point to the same object and would print out the same value.

The exam won't always make the code easy to read by having only one method per line. What do you think this example prints?

```
4: StringBuilder a = new StringBuilder("abc");
5: StringBuilder b = a.append("de");
6: b = b.append("f").append("g");
7: System.out.println("a=" + a);
8: System.out.println("b=" + b);
```

Did you say both print `"abcdefg"`? Good. There's only one `StringBuilder` object here. We know that because `new StringBuilder()` was called only once. On line 5, there are two variables referring to that object, which has a value of `"abcde"`. On line 6, those two variables are still referring to that same object, which now has a value of `"abcdefg"`. Incidentally, the assignment back to `b` does absolutely nothing. `b` is already pointing to that `StringBuilder`.

Creating a *StringBuilder*

There are three ways to construct a `StringBuilder`:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
```

The first says to create a `StringBuilder` containing an empty sequence of characters and assign `sb1` to point to it. The second says to create a `StringBuilder` containing a specific value and assign `sb2` to point to it. For the first two, it tells Java to manage the implementation details. The final example tells Java that we have some idea of how big the eventual value will be and would like the `StringBuilder` to reserve a certain capacity, or number of slots, for characters.

Important *StringBuilder* Methods

As with `String`, we aren't going to cover every single method in the `StringBuilder` class. These are the ones you might see on the exam.

charAt(), *indexOf()*, *length()*, and *substring()*

These four methods work exactly the same as in the `String` class. Be sure you can identify the output of this example:

```
StringBuilder sb = new StringBuilder("animals");
String sub = sb.substring(sb.indexOf("a"), sb.indexOf("al"));
int len = sb.length();
char ch = sb.charAt(6);
System.out.println(sub + " " + len + " " + ch);
```

The correct answer is `anim 7 s`. The `indexOf()` method calls return 0 and 4, respectively. `substring()` returns the `String` starting with index 0 and ending right before index 4.

`length()` returns 7 because it is the number of characters in the `StringBuilder` rather than an index. Finally, `charAt()` returns the character at index 6. Here we do start with 0 because we are referring to indexes. If any of this doesn't sound familiar, go back and read the section on `String` again.

Notice that `substring()` returns a `String` rather than a `StringBuilder`. That is why `sb` is not changed. `substring()` is really just a method that inquires about what the state of the `StringBuilder` happens to be.

append()

The `append()` method is by far the most frequently used method in `StringBuilder`. In fact, it is so frequently used that we just started using it without comment. Luckily, this method does just what it sounds like: It adds the parameter to the `StringBuilder` and returns a reference to the current `StringBuilder`. One of the method signatures is as follows:

```
StringBuilder append(String str)
```

Notice that we said *one* of the method signatures. There are more than 10 method signatures that look similar but that take different data types as parameters. All those methods are provided so you can write code like this:

```
StringBuilder sb = new StringBuilder().append(1).append('c');
sb.append("-").append(true);
System.out.println(sb);      // 1c-true
```

Nice method chaining, isn't it? `append()` is called directly after the constructor. By having all these method signatures, you can just call `append()` without having to convert your parameter to a `String` first.

insert()

The `insert()` method adds characters to the `StringBuilder` at the requested index and returns a reference to the current `StringBuilder`. Just like `append()`, there are lots of method signatures for different types. Here's one:

```
StringBuilder insert(int offset, String str)
```

Pay attention to the offset in these examples. It is the index where we want to insert the requested parameter.

```
3: StringBuilder sb = new StringBuilder("animals");
4: sb.insert(7, "-");                  // sb = animals-
5: sb.insert(0, "-");                  // sb = -animals-
6: sb.insert(4, "-");                  // sb = -ani-mals-
7: System.out.println(sb);
```

Line 4 says to insert a dash at index 7, which happens to be the end of the sequence of characters. Line 5 says to insert a dash at index 0, which happens to be the very beginning. Finally, line 6 says to insert a dash right before index 4. The exam creators will try to trip you up on this. As we add and remove characters, their indexes change. When you see a question dealing with such operations, draw what is going on so you won't be confused.

delete() and *deleteCharAt()*

The `delete()` method is the opposite of the `insert()` method. It removes characters from the sequence and returns a reference to the current `StringBuilder`. The `deleteCharAt()` method is convenient when you want to delete only one character. The method signatures are as follows:

```
StringBuilder delete(int startIndex, int endIndex)
StringBuilder deleteCharAt(int index)
```

The following code shows how to use these methods:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 3);                  // sb = adef
sb.deleteCharAt(5);               // throws an exception
```

First, we delete the characters starting with index 1 and ending right before index 3. This gives us `adef`. Next, we ask Java to delete the character at position 5. However, the remaining value is only four characters long, so it throws a `StringIndexOutOfBoundsException`.

The `delete()` method is more flexible than some others when it comes to array indexes. If you specify a second parameter that is past the end of the `StringBuilder`, Java will just assume you meant the end. That means this code is legal:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 100);                  // sb = a
```

replace()

The `replace()` method works differently for `StringBuilder` than it did for `String`. The method signature is as follows:

```
StringBuilder replace(int startIndex, int endIndex, String newString)
```

The following code shows how to use this method:

```
StringBuilder builder = new StringBuilder("pigeon dirty");
builder.replace(3, 6, "sty");
System.out.println(builder);  // pigsty dirty
```

First, Java deletes the characters starting with index 3 and ending right before index 6. This gives us `pig dirty`. Then Java inserts to the value `"sty"` in that position.

In this example, the number of characters removed and inserted is the same. However, there is no reason that it has to be. What do you think this does?

```
StringBuilder builder = new StringBuilder("pigeon dirty");
builder.replace(3, 100, "");
System.out.println(builder);
```

It actually prints `"pig"`. Remember the method is first doing a logical delete. The `replace()` method allows specifying a second parameter that is past the end of the `StringBuilder`. That means only the first three characters remain.

reverse()

After all that, it's time for a nice, easy method. The `reverse()` method does just what it sounds like: it reverses the characters in the sequences and returns a reference to the current `StringBuilder`. The method signature is as follows:

```
StringBuilder reverse()
```

The following code shows how to use this method:

```
StringBuilder sb = new StringBuilder("ABC");
sb.reverse();
System.out.println(sb);
```

As expected, this prints `CBA`. This method isn't that interesting. Maybe the exam creators like to include it to encourage you to write down the value rather than relying on memory for indexes.

toString()

The last method converts a `StringBuilder` into a `String`. The method signature is as follows:

```
String toString()
```

The following code shows how to use this method:

```
StringBuilder sb = new StringBuilder("ABC");
String s = sb.toString();
```

Often `StringBuilder` is used internally for performance purposes, but the end result needs to be a `String`. For example, maybe it needs to be passed to another method that is expecting a `String`.

Understanding Equality

In Chapter 3, you learned how to use `==` to compare numbers and that object references refer to the same object. In this section, we will look at what it means for two objects to be equivalent or the same. We will also look at the impact of the `String` pool on equality.

Comparing *equals()* and ==

Consider the following code that uses `==` with objects:

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
```

```
System.out.println(one == two); // false
System.out.println(one == three); // true
```

Since this example isn't dealing with primitives, we know to look for whether the references are referring to the same object. `one` and `two` are both completely separate `StringBuilder` objects, giving us two objects. Therefore, the first print statement gives us `false`. `three` is more interesting. Remember how `StringBuilder` methods like to return the current reference for chaining? This means `one` and `three` both point to the same object, and the second print statement gives us `true`.

You saw earlier that you can say you want logical equality rather than object equality for `String` objects:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x.equals(z)); // true
```

This works because the authors of the `String` class implemented a standard method called `equals` to check the values inside the `String` rather than the string reference itself. If a class doesn't have an `equals` method, Java determines whether the references point to the same object—which is exactly what `==` does.

In case you are wondering, the authors of `StringBuilder` did not implement `equals()`. If you call `equals()` on two `StringBuilder` instances, it will check reference equality. You can call `toString()` on `StringBuilder` to get a `String` to check for equality instead.

The exam will test you on your understanding of equality with objects they define too. For example, the following `Tiger` class works just like `StringBuilder` but is easier to understand:

```
1:  public class Tiger {
2:     String name;
```

```
3:     public static void main(String[] args) {
4:        Tiger t1 = new Tiger();
5:        Tiger t2 = new Tiger();
6:        Tiger t3 = t1;
7:        System.out.println(t1 == t3);      // true
8:        System.out.println(t1 == t2);      // false
9:        System.out.println(t1.equals(t2)); // false
10: } }
```

The first two statements check object reference equality. Line 7 prints `true` because we are comparing references to the same object. Line 8 prints `false` because the two object references are different. Line 9 prints `false` since `Tiger` does not implement `equals()`. Don't worry—you aren't expected to know how to implement `equals()` for this exam.

Finally, the exam might try to trick you with a question like this. Can you guess why the code doesn't compile?

```
String string = "a";
StringBuilder builder = new StringBuilder("a");
System.out.println(string == builder); //DOES NOT COMPILE
```

Remember that `==` is checking for object reference equality. The compiler is smart enough to know that two references can't possibly point to the same object when they are completely different types.

The *String* Pool

Since strings are everywhere in Java, they use up a lot of memory. In some production applications, they can use a large amount of memory in the entire program. Java realizes that many strings repeat in the program and solves this issue by reusing common ones. The *string pool*, also known as the intern pool, is a location in the Java virtual machine (JVM) that collects all these strings.

The string pool contains literal values and constants that appear in your program. For example, `"name"` is a literal and therefore goes into the string pool. `myObject.toString()` is a string but not a literal, so it does not go into the string pool.

Let's now visit the more complex and confusing scenario, `String` equality, made so in part because of the way the JVM reuses `String` literals.

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y);    // true
```

Remember that `String`s are immutable and literals are pooled. The JVM created only one literal in memory. `x` and `y` both point to the same location in memory; therefore, the statement outputs `true`. It gets even trickier. Consider this code:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x == z); // false
```

In this example, we don't have two of the same `String` literal. Although `x` and `z` happen to evaluate to the same string, one is computed at runtime. Since it isn't the same at compile-time, a new `String` object is created. Let's try another one. What do you think is output here?

```
String singleString = "hello world";
String concat = "hello ";
concat += "world";
System.out.println(singleString == concat);
```

This prints `false`. Concatenation is just like calling a method and results in a new `String`. You can even force the issue by creating a new `String`:

```
String x = "Hello World";
String y = new String("Hello World");
System.out.println(x == y); // false
```

The former says to use the string pool normally. The second says "No, JVM, I really don't want you to use the string pool. Please create a new object for me even though it is less efficient."

You can also do the opposite and tell Java to use the string pool. The `intern()` method will use an object in the string pool if one is present. If the literal is not yet in the string pool, Java will add it at this time.

```
String name = "Hello World";
String name2 = new String("Hello World").intern();
System.out.println(name == name2);     // true
```

First we tell Java to use the string pool normally for `name`. Then for `name2`, we tell Java to create a new object using the constructor but to intern it and use the string pool anyway. Since both variables point to the same reference in the string pool, we can use the `==` operator.

Let's try another one. What do you think this prints out? Be careful. It is tricky.

```
15: String first = "rat" + 1;
16: String second = "r" + "a" + "t" + "1";
17: String third = "r" + "a" + "t" + new String("1");
18: System.out.println(first == second);
19: System.out.println(first == second.intern());
20: System.out.println(first == third);
21: System.out.println(first == third.intern());
```

On line 15, we have a compile-time constant that automatically gets placed in the string pool as `"rat1"`. On line 16, we have a more complicated expression that is also a compile-time constant. Therefore, `first`

and `second` share the same string pool reference. This makes line 18 and 19 print `true`.

On line 17, we have a `String` constructor. This means we no longer have a compile-time constant, and `third` does not point to a reference in the string pool. Therefore, line 20 prints `false`. On line 21, the `intern()` call looks in the string pool. Java notices that `first` points to the same `String` and prints `true`.

When you write programs, you wouldn't want to create a `String` of a `String` or use the `intern()` method. For the exam, you need to know that both are allowed and how they behave.

Remember to never use `intern()` or `==` to compare `String` objects in your code. The only time you should have to deal with these is on the exam.

Understanding Java Arrays

Up to now, we've been referring to the `String` and `StringBuilder` classes as a "sequence of characters." This is true. They are implemented using an *array* of characters. An array is an area of memory on the heap with space for a designated number of elements. A `String` is implemented as an array with some methods that you might want to use when dealing with characters specifically. A `StringBuilder` is implemented as an array where the array object is replaced with a new bigger array object when it runs out of space to store all the characters. A big difference is that an array can be of any other Java type. If we didn't want to use a `String` for some reason, we could use an array of `char` primitives directly:

```
char[] letters;
```

This wouldn't be very convenient because we'd lose all the special properties `String` gives us, such as writing `"Java"`. Keep in mind that `letters` is a reference variable and not a primitive. `char` is a primitive. But `char` is what goes into the array and not the type of the array itself. The array itself is of type `char[]`. You can mentally read the brackets (`[]`) as "array."

In other words, an array is an ordered list. It can contain duplicates. In this section, we'll look at creating an array of primitives and objects, sorting, searching, varargs, and multidimensional arrays.

Creating an Array of Primitives

The most common way to create an array looks like this:

```
int[] numbers1 = new int[3];
```

The basic parts are shown in Figure 5.3. It specifies the type of the array (`int`) and the size (3). The brackets tell you this is an array.

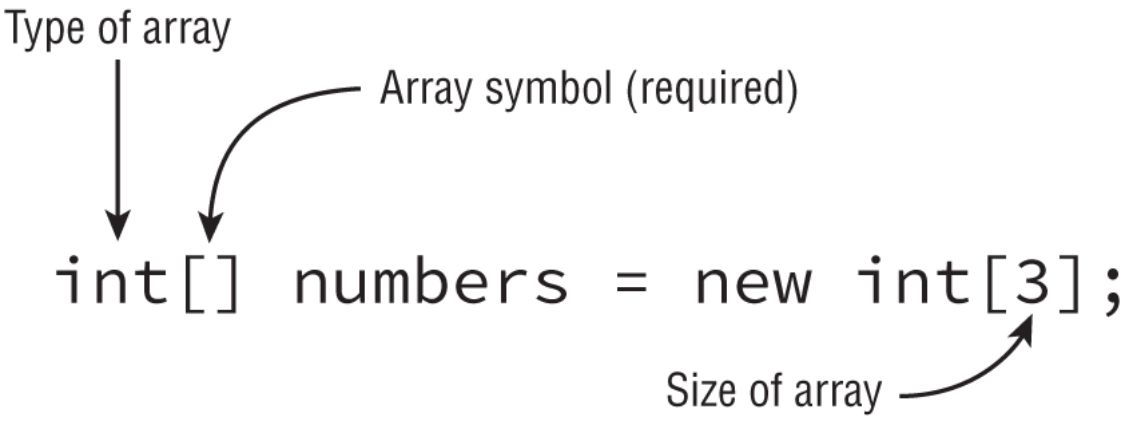

FIGURE 5.3 The basic structure of an array

When you use this form to instantiate an array, all elements are set to the default value for that type. As you learned in Chapter 2, the default value

of an `int` is 0. Since `numbers1` is a reference variable, it points to the array object, as shown in Figure 5.4. As you can see, the default value for all the elements is 0. Also, the indexes start with 0 and count up, just as they did for a `String`.

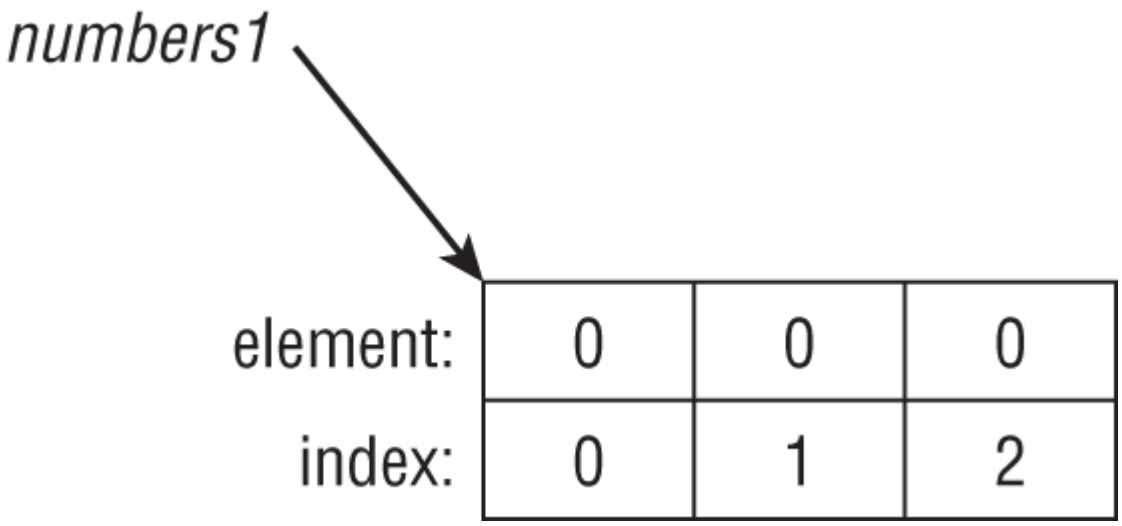

FIGURE 5.4 An empty array

Another way to create an array is to specify all the elements it should start out with:

```
int[] numbers2 = new int[] {42, 55, 99};
```

In this example, we also create an `int` array of size 3. This time, we specify the initial values of those three elements instead of using the defaults. Figure 5.5 shows what this array looks like.

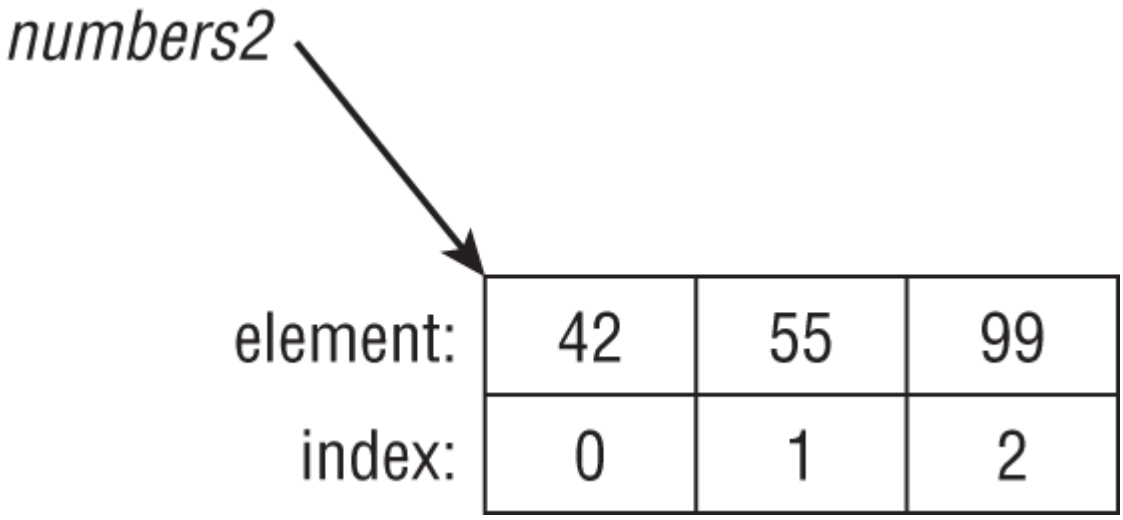

FIGURE 5.5 An initialized array

Java recognizes that this expression is redundant. Since you are specifying the type of the array on the left side of the equal sign, Java already knows the type. And since you are specifying the initial values, it already knows the size. As a shortcut, Java lets you write this:

```
int[] numbers2 = {42, 55, 99};
```

This approach is called an *anonymous array*. It is anonymous because you don't specify the type and size.

Finally, you can type the [] before or after the name, and adding a space is optional. This means that all five of these statements do the exact same thing:

```
int[] numAnimals;
int [] numAnimals2;
int []numAnimals3;
int numAnimals4[];
int numAnimals5 [];
```

Most people use the first one. You could see any of these on the exam, though, so get used to seeing the brackets in odd places.

MULTIPLE "ARRAYS" IN DECLARATIONS

What types of reference variables do you think the following code creates?

```
int[] ids, types;
```

The correct answer is two variables of type `int[]`. This seems logical enough. After all, `int a, b;` created two `int` variables. What about this example?

```
int ids[], types;
```

All we did was move the brackets, but it changed the behavior. This time we get one variable of type `int[]` and one variable of type `int`. Java sees this line of code and thinks something like this: "They want two variables of type `int`. The first one is called `ids[]`. This one is an `int[]` called `ids`. The second one is just called `types`. No brackets, so it is a regular integer."

Needless to say, you shouldn't write code that looks like this. But you do need to understand it for the exam.

Creating an Array with Reference Variables

You can choose any Java type to be the type of the array. This includes classes you create yourself. Let's take a look at a built-in type with `String`:

```
public class ArrayType {
   public static void main(String args[]) {
      String [] bugs = { "cricket", "beetle", "ladybug" };
      String [] alias = bugs;
      System.out.println(bugs.equals(alias));     // true
```

```
        System.out.println(
           bugs.toString()); //[Ljava.lang.String;@160bc7c0
  } }
```

We can call `equals()` because an array is an object. It returns `true` because of reference equality. The `equals()` method on arrays does not look at the elements of the array. Remember, this would work even on an `int[]` too. `int` is a primitive; `int[]` is an object.

The second print statement is even more interesting. What on earth is `[Ljava.lang .String;@160bc7c0` ? You don't have to know this for the exam, but `[L` means it is an array, `java.lang.String` is the reference type, and `160bc7c0` is the hash code. You'll get different numbers and letters each time you run it since this is a reference.

Since Java 5, Java has provided a method that prints an array nicely: `Arrays.toString(bugs)` would print `[cricket, beetle, ladybug]`.

Make sure you understand Figure 5.6. The array does not allocate space for the `String` objects. Instead, it allocates space for a reference to where the objects are really stored.

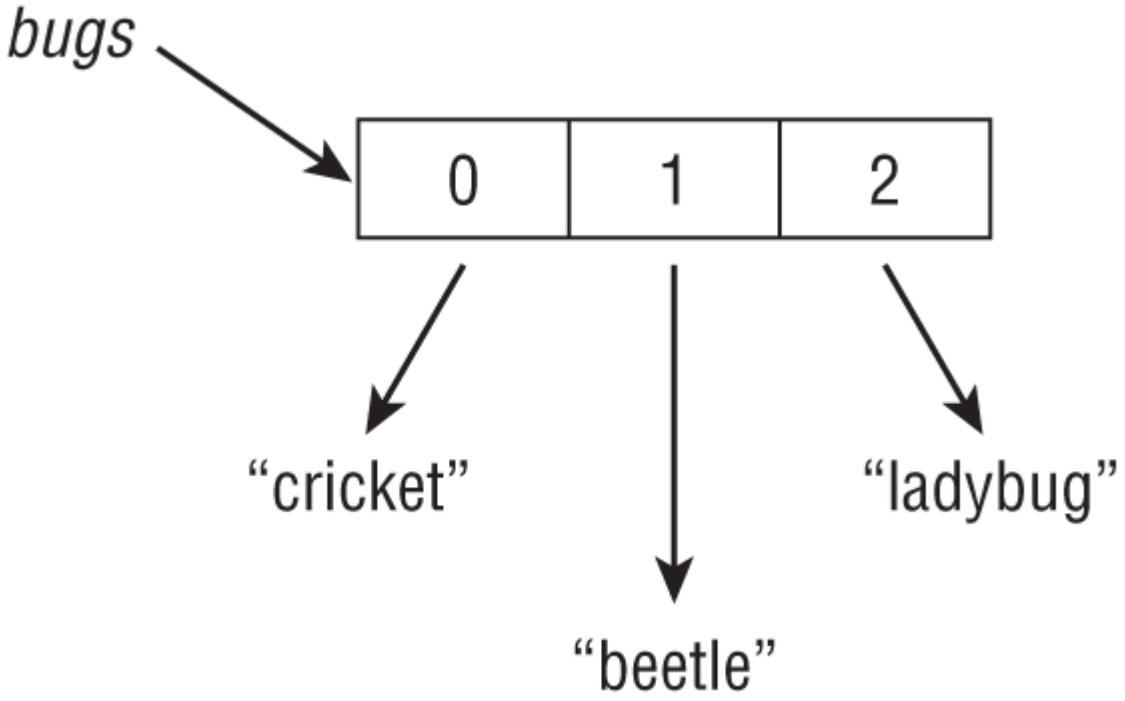

FIGURE 5.6 An array pointing to strings

As a quick review, what do you think this array points to?

```
class Names {
   String names[];
}
```

You got us. It was a review of Chapter 2 and not our discussion on arrays. The answer is `null`. The code never instantiated the array, so it is just a reference variable to `null`. Let's try that again—what do you think this array points to?

```
class Names {
   String names[] = new String[2];
}
```

It is an array because it has brackets. It is an array of type `String` since that is the type mentioned in the declaration. It has two elements because the length is 2. Each of those two slots currently is `null` but has the potential to point to a `String` object.

Remember casting from the previous chapter when you wanted to force a bigger type into a smaller type? You can do that with arrays too:

```
3: String[] strings = { "stringValue" };
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
6: againStrings[0] = new StringBuilder();   // DOES NOT COMPILE
7: objects[0] = new StringBuilder();        // careful!
```

Line 3 creates an array of type `String`. Line 4 doesn't require a cast because `Object` is a broader type than `String`. On line 5, a cast is needed because we are moving to a more specific type. Line 6 doesn't compile because a `String[]` only allows `String` objects and `StringBuilder` is not a `String`.

Line 7 is where this gets interesting. From the point of view of the compiler, this is just fine. A `StringBuilder` object can clearly go in an `Object[]`. The problem is that we don't actually have an `Object[]`. We have a `String[]` referred to from an `Object[]` variable. At runtime, the code throws an `ArrayStoreException`. You don't need to memorize the name of this exception, but you do need to know that the code will throw an exception.

Using an Array

Now that you know how to create an array, let's try accessing one:

```
4: String[] mammals = {"monkey", "chimp", "donkey"};
5: System.out.println(mammals.length);          // 3
6: System.out.println(mammals[0]);              // monkey
7: System.out.println(mammals[1]);              // chimp
8: System.out.println(mammals[2]);              // donkey
```

Line 4 declares and initializes the array. Line 5 tells us how many elements the array can hold. The rest of the code prints the array. Notice ele-

ments are indexed starting with 0. This should be familiar from `String` and `StringBuilder`, which also start counting with 0. Those classes also counted `length` as the number of elements. Note that there are no parentheses after `length` since it is not a method.

To make sure you understand how `length` works, what do you think this prints?

```
String[] birds = new String[6];
System.out.println(birds.length);
```

The answer is `6`. Even though all six elements of the array are `null`, there are still six of them. `length` does not consider what is in the array; it only considers how many slots have been allocated.

It is very common to use a loop when reading from or writing to an array. This loop sets each element of `numbers` to five higher than the current index:

```
5: int[] numbers = new int[10];
6: for (int i = 0; i < numbers.length; i++)
7:    numbers[i] = i + 5;
```

Line 5 simply instantiates an array with 10 slots. Line 6 is a `for` loop that uses an extremely common pattern. It starts at index 0, which is where an array begins as well. It keeps going, one at a time, until it hits the end of the array. Line 7 sets the current element of `numbers`.

The exam will test whether you are being observant by trying to access elements that are not in the array. Can you tell why each of these throws an `ArrayIndexOutOfBoundsException` for our array of size 10?

```
numbers[10] = 3;
numbers[numbers.length] = 5;
for (int i = 0; i <= numbers.length; i++) numbers[i] = i + 5;
```

The first one is trying to see whether you know that indexes start with 0. Since we have 10 elements in our array, this means only `numbers[0]` through `numbers[9]` are valid. The second example assumes you are clever enough to know 10 is invalid and disguises it by using the `length` field. However, the length is always one more than the maximum valid index. Finally, the `for` loop incorrectly uses `<=` instead of `<`, which is also a way of referring to that 10th element.

Sorting

Java makes it easy to sort an array by providing a sort method—or rather, a bunch of sort methods. Just like `StringBuilder` allowed you to pass almost anything to `append()`, you can pass almost any array to `Arrays.sort()`.

`Arrays` is the first class provided by Java we have used that requires an import. To use it, you must have either of the following two statements in your class:

```
import java.util.*;          // import whole package including Arrays
import java.util.Arrays;     // import just Arrays
```

There is one exception, although it doesn't come up often on the exam. You can write `java.util.Arrays` every time it is used in the class instead of specifying it as an import.

Remember that if you are shown a code snippet with a line number that doesn't begin with 1, you can assume the necessary imports are there. Similarly, you can assume the imports are present if you are shown a snippet of a method.

This simple example sorts three numbers:

```
int[] numbers = { 6, 9, 1 };
Arrays.sort(numbers);
for (int i = 0; i < numbers.length; i++)
   System.out.print(numbers[i] +  " ");
```

The result is `1 6 9`, as you should expect it to be. Notice that we looped through the output to print the values in the array. Just printing the array variable directly would give the annoying hash of `[I@2bd9c3e7`. Alternatively, we could have printed `Arrays .toString(numbers)` instead of using the loop. That would have output `[1, 6, 9]`.

Try this again with `String` types:

```
String[] strings = { "10", "9", "100" };
Arrays.sort(strings);
for (String string : strings)
   System.out.print(string + " ");
```

This time the result might not be what you expect. This code outputs `10 100 9`. The problem is that `String` sorts in alphabetic order, and 1 sorts before 9. (Numbers sort before letters, and uppercase sorts before lowercase, in case you were wondering.) For the 1Z0-816 exam, you'll learn how to create custom sort orders using something called a *comparator*.

Did you notice we snuck in the enhanced `for` loop in this example? Since we aren't using the index, we don't need the traditional `for` loop. That won't stop the exam creators from using it, though, so we'll be sure to use both to keep you sharp!

Searching

Java also provides a convenient way to search—but only if the array is already sorted. Table 5.1 covers the rules for binary search.

TABLE 5.1 Binary search rules

Scenario	Result
Target element found in sorted array	Index of match
Target element not found in sorted array	Negative value showing one smaller than the negative of the index, where a match needs to be inserted to preserve sorted order
Unsorted array	A surprise—this result isn't predictable

Let's try these rules with an example:

```
3: int[] numbers = {2,4,6,8};
4: System.out.println(Arrays.binarySearch(numbers, 2)); // 0
5: System.out.println(Arrays.binarySearch(numbers, 4)); // 1
6: System.out.println(Arrays.binarySearch(numbers, 1)); // -1
7: System.out.println(Arrays.binarySearch(numbers, 3)); // -2
8: System.out.println(Arrays.binarySearch(numbers, 9)); // -5
```

Take note of the fact that line 3 is a sorted array. If it wasn't, we couldn't apply either of the other rules. Line 4 searches for the index of 2. The answer is index 0. Line 5 searches for the index of 4, which is 1.

Line 6 searches for the index of 1. Although 1 isn't in the list, the search can determine that it should be inserted at element 0 to preserve the sorted order. Since 0 already means something for array indexes, Java needs to subtract 1 to give us the answer of –1. Line 7 is similar. Although 3 isn't in the list, it would need to be inserted at element 1 to preserve the

sorted order. We negate and subtract 1 for consistency, getting –1 –1, also known as –2. Finally, line 8 wants to tell us that 9 should be inserted at index 4. We again negate and subtract 1, getting –4 –1, also known as –5.

What do you think happens in this example?

```
5: int[] numbers = new int[] {3,2,1};
6: System.out.println(Arrays.binarySearch(numbers, 2));
7: System.out.println(Arrays.binarySearch(numbers, 3));
```

Note that on line 5, the array isn't sorted. This means the output will not be predictable. When testing this example, line 6 correctly gave 1 as the output. However, line 7 gave the wrong answer. The exam creators will not expect you to know what incorrect values come out. As soon as you see the array isn't sorted, look for an answer choice about unpredictable output.

On the exam, you need to know what a binary search returns in various scenarios. Oddly, you don't need to know why "binary" is in the name. In case you are curious, a binary search splits the array into two equal pieces (remember 2 is binary) and determines which half the target is in. It repeats this process until only one element is left.

Comparing

Java also provides methods to compare two arrays to determine which is "smaller." First we will cover the `compare()` method and then go on to `mismatch()`.

compare()

There are a bunch of rules you need to know before calling `compare()`. Luckily, these are the same rules you'll need to know for the 1Z0-816 exam when writing a `Comparator`.

First you need to learn what the return value means. You do not need to know the exact return values, but you do need to know the following:

- A negative number means the first array is smaller than the second.
- A zero means the arrays are equal.
- A positive number means the first array is larger than the second.

Here's an example:

```
System.out.println(Arrays.compare(new int[] {1}, new int[] {2}));
```

This code prints a negative number. It should be pretty intuitive that 1 is smaller than 2, making the first array smaller.

Now that you know how to compare a single value, let's look at how to compare arrays of different lengths:

- If both arrays are the same length and have the same values in each spot in the same order, return zero.
- If all the elements are the same but the second array has extra elements at the end, return a negative number.
- If all the elements are the same but the first array has extra elements at the end, return a positive number.
- If the first element that differs is smaller in the first array, return a negative number.
- If the first element that differs is larger in the first array, return a positive number.

Finally, what does smaller mean? Here are some more rules that apply here and to `compareTo()`, which you'll see in Chapter 6, "Lambdas and Functional Interfaces":

- `null` is smaller than any other value.
- For numbers, normal numeric order applies.
- For strings, one is smaller if it is a prefix of another.

- For strings/characters, numbers are smaller than letters.
- For strings/characters, uppercase is smaller than lowercase.

Table 5.2 shows examples of these rules in action.

TABLE 5.2 `Arrays.compare()` examples

First array	Second array	Result	Reason
`new int[] {1, 2}`	`new int[] {1}`	Positive number	The first element is the same, but the first array is longer.
`new int[] {1, 2}`	`new int[] {1, 2}`	Zero	Exact match
`new String[] {"a"}`	`new String[] {"aa"}`	Negative number	The first element is a substring of the second.
`new String[] {"a"}`	`new String[] {"A"}`	Positive number	Uppercase is smaller than lowercase.
`new String[] {"a"}`	`new String[] {null}`	Positive number	null is smaller than a letter.

Finally, this code does not compile because the types are different. When comparing two arrays, they must be the same array type.

```
System.out.println(Arrays.compare(
   new int[] {1}, new String[] {"a"})); // DOES NOT COMPILE
```

mismatch()

Now that you are familiar with `compare()`, it is time to learn about `mismatch()`. If the arrays are equal, `mismatch()` returns `-1`. Otherwise, it returns the first index where they differ. Can you figure out what these print?

```
System.out.println(Arrays.mismatch(new int[] {1}, new int[] {1}));
System.out.println(Arrays.mismatch(new String[] {"a"},
   new String[] {"A"}));
System.out.println(Arrays.mismatch(new int[] {1, 2}, new int[] {1}));
```

In the first example, the arrays are the same, so the result is `-1`. In the second example, the entries at element 0 are not equal, so the result is `0`. In the third example, the entries at element 0 are equal, so we keep looking. The element at index 1 is not equal. Or more specifically, one array has an element at index 1, and the other does not. Therefore, the result is `1`.

To make sure you understand the `compare()` and `mismatch()` methods, study Table 5.3. If you don't understand why all of the values are there, please go back and study this section again.

TABLE 5.3 Equality vs. comparison vs. mismatch

Method	When arrays are the same	When arrays are different
`equals()`	`true`	`false`
`compare()`	`0`	Positive or negative number
`mismatch()`	`-1`	Zero or positive index

Varargs

When you're creating an array yourself, it looks like what we've seen thus far. When one is passed to your method, there is another way it can look. Here are three examples with a `main()` method:

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

The third example uses a syntax called *varargs* (variable arguments), which you saw in Chapter 1, "Welcome to Java." You'll learn how to call a method using varargs in Chapter 7, "Methods and Encapsulation." For now, all you need to know is that you can use a variable defined using varargs as if it were a normal array. For example, `args.length` and `args[0]` are legal.

Multidimensional Arrays

Arrays are objects, and of course array components can be objects. It doesn't take much time, rubbing those two facts together, to wonder whether arrays can hold other arrays, and of course they can.

Creating a Multidimensional Array

Multiple array separators are all it takes to declare arrays with multiple dimensions. You can locate them with the type or variable name in the declaration, just as before:

```
int[][] vars1;          // 2D array
int vars2 [][];         // 2D array
int[] vars3[];          // 2D array
int[] vars4 [], space [][];  // a 2D AND a 3D array
```

The first two examples are nothing surprising and declare a two-dimensional (2D) array. The third example also declares a 2D array. There's no good reason to use this style other than to confuse readers with your code. The final example declares two arrays on the same line. Adding up the brackets, we see that the `vars4` is a 2D array and `space` is a 3D array. Again, there's no reason to use this style other than to confuse readers of your code. The exam creators like to try to confuse you, though. Luckily, you are on to them and won't let this happen to you!

You can specify the size of your multidimensional array in the declaration if you like:

```
String [][] rectangle = new String[3][2];
```

The result of this statement is an array rectangle with three elements, each of which refers to an array of two elements. You can think of the addressable range as [0][0] through [2][1], but don't think of it as a structure of addresses like [0,0] or [2,1].

Now suppose we set one of these values:

```
rectangle[0][1] = "set";
```

You can visualize the result as shown in Figure 5.7. This array is sparsely populated because it has a lot of `null` values. You can see that `rectangle` still points to an array of three elements and that we have three arrays of two elements. You can also follow the trail from reference to the one value pointing to a `String`. First you start at index 0 in the top array. Then you go to index 1 in the next array.

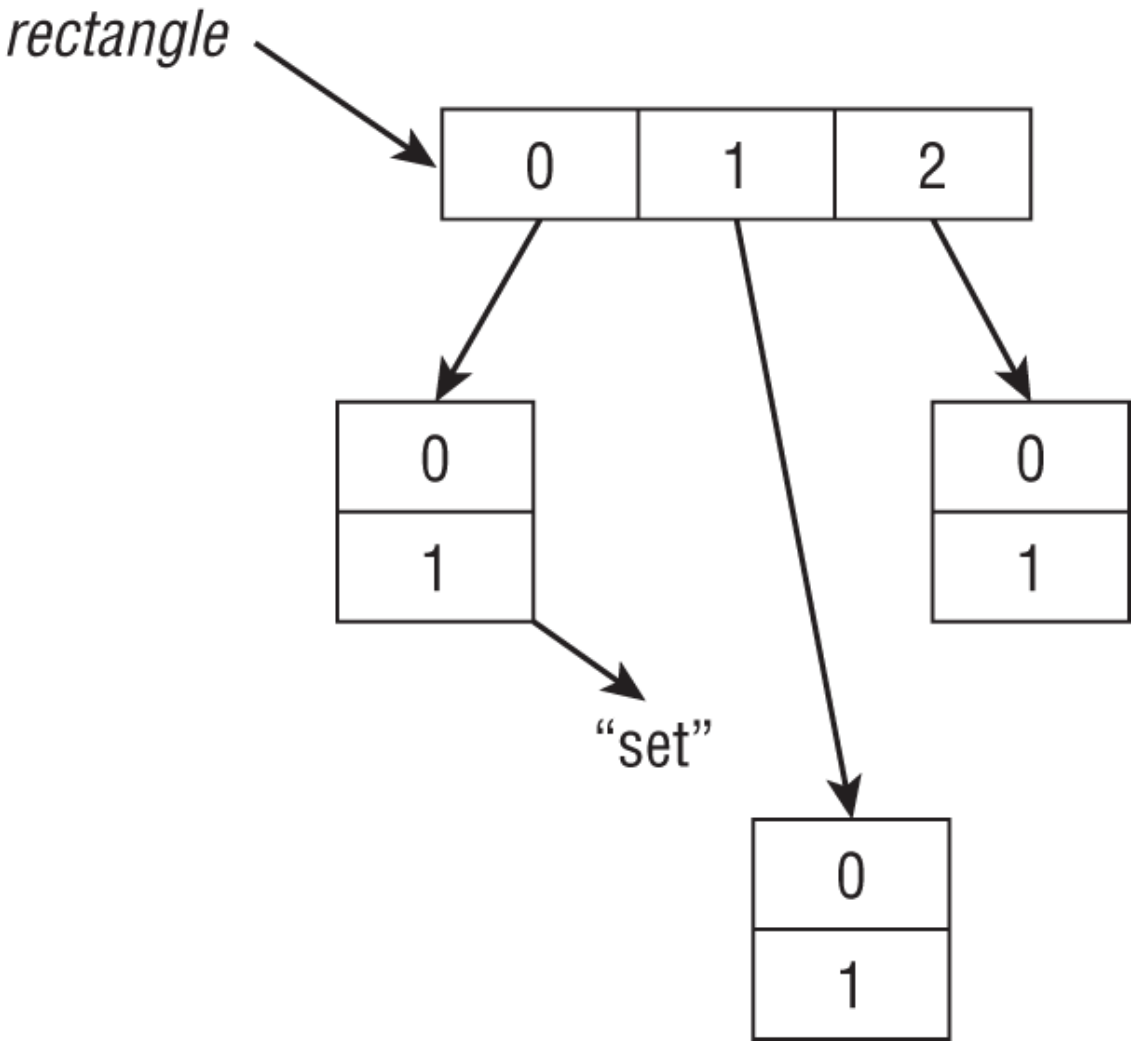

FIGURE 5.7 A sparsely populated multidimensional array

While that array happens to be rectangular in shape, an array doesn't need to be. Consider this one:

```
int[][] differentSizes = {{1, 4}, {3}, {9,8,7}};
```

We still start with an array of three elements. However, this time the elements in the next level are all different sizes. One is of length 2, the next length 1, and the last length 3 (see Figure 5.8). This time the array is of primitives, so they are shown as if they are in the array themselves.

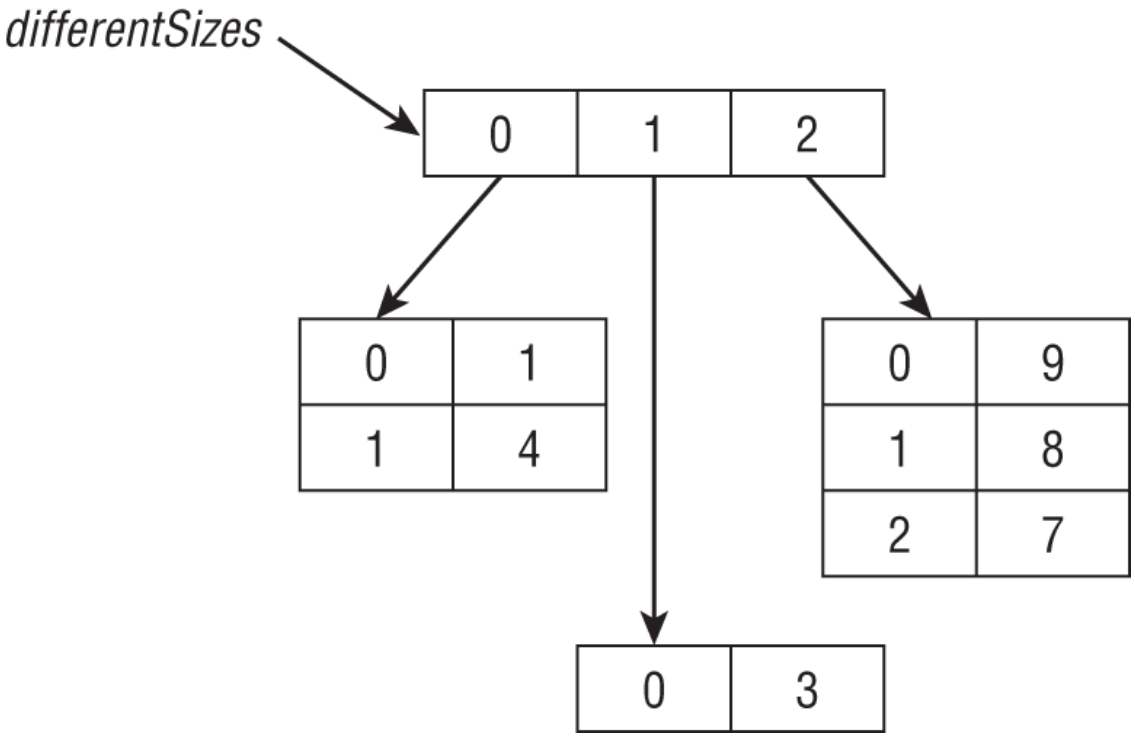

FIGURE 5.8 An asymmetric multidimensional array

Another way to create an asymmetric array is to initialize just an array's first dimension and define the size of each array component in a separate statement:

```
int [][] args = new int[4][];
args[0] = new int[5];
args[1] = new int[3];
```

This technique reveals what you really get with Java: arrays of arrays that, properly managed, offer a multidimensional effect.

Using a Multidimensional Array

The most common operation on a multidimensional array is to loop through it. This example prints out a 2D array:

```
int[][] twoD = new int[3][2];
for (int i = 0; i < twoD.length; i++) {
   for (int j = 0; j < twoD[i].length; j++)
      System.out.print(twoD[i][j] + " "); // print element
   System.out.println();                  // time for a new row
}
```

We have two loops here. The first uses index `i` and goes through the first subarray for `twoD`. The second uses a different loop variable `j`. It is important that these be different variable names so the loops don't get mixed up. The inner loop looks at how many elements are in the second-level array. The inner loop prints the element and leaves a space for readability. When the inner loop completes, the outer loop goes to a new line and repeats the process for the next element.

This entire exercise would be easier to read with the enhanced `for` loop.

```
for (int[] inner : twoD) {
   for (int num : inner)
      System.out.print(num + " ");
   System.out.println();
}
```

We'll grant you that it isn't fewer lines, but each line is less complex, and there aren't any loop variables or terminating conditions to mix up.

Understanding an *ArrayList*

An array has one glaring shortcoming: You have to know how many elements will be in the array when you create it, and then you are stuck with that choice. Just like a `StringBuilder`, an `ArrayList` can change capac-

ity at runtime as needed. Like an array, an `ArrayList` is an ordered sequence that allows duplicates.

As when we used `Arrays.sort`, `ArrayList` requires an import. To use it, you must have either of the following two statements in your class:

```
import java.util.*;          // import whole package
import java.util.ArrayList; // import just ArrayList
```

In this section, we'll look at creating an `ArrayList`, common methods, autoboxing, conversion, and sorting.

Experienced programmers, take note: This section is simplified and doesn't cover a number of topics that are out of scope for this exam.

Creating an *ArrayList*

As with `StringBuilder`, there are three ways to create an `ArrayList`:

```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);
```

The first says to create an `ArrayList` containing space for the default number of elements but not to fill any slots yet. The second says to create an `ArrayList` containing a specific number of slots, but again not to assign any. The final example tells Java that we want to make a copy of another `ArrayList`. We copy both the size and contents of that `ArrayList`. Granted, `list2` is empty in this example, so it isn't particularly interesting.

Although these are the only three constructors you need to know, you do need to learn some variants of it. The previous examples were the old pre–Java 5 way of creating an `ArrayList`. They still work, and you still need to know they work. You also need to know the new and improved

way. Java 5 introduced *generics*, which allow you to specify the type of class that the `ArrayList` will contain.

```
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>();
```

Java 5 allows you to tell the compiler what the type would be by specifying it between `<` and `>`. Starting in Java 7, you can even omit that type from the right side. The `<` and `>` are still required, though. This is called the *diamond operator* because `<>` looks like a diamond.

USING VAR WITH ARRAYLIST

Now that `var` can be used to obscure data types, there is a whole new group of questions that can be asked with generics. Consider this code mixing the two:

```
var strings = new ArrayList<String>();
strings.add("a");
for (String s: strings) {  }
```

The type of `var` is `ArrayList<String>`. This means you can add a `String` or loop through the `String` objects. What if we use the diamond operator with `var`?

```
var list = new ArrayList<>();
```

Believe it or not, this does compile. The type of the `var` is `ArrayList<Object>`. Since there isn't a type specified for the generic, Java has to assume the ultimate superclass. This is a bit silly and unexpected, so please don't write this. But if you see it on the exam, you'll know what to expect. Now can you figure out why this doesn't compile?

```
var list = new ArrayList<>();
list.add("a");
for (String s: list) { } // DOES NOT COMPILE
```

The type of `var` is `ArrayList<Object>`. Since there isn't a type in the diamond operator, Java has to assume the most generic option it can. Therefore, it picks `Object`, the ultimate superclass. Adding a `String` to the list is fine. You can add any subclass of `Object`. However, in the loop, we need to use the `Object` type rather than `String`.

Just when you thought you knew everything about creating an `ArrayList`, there is one more thing you need to know. `ArrayList` implements an interface called `List`. In other words, an `ArrayList` is a `List`. You will learn about interfaces later in the book. In the meantime, just know that you can store an `ArrayList` in a `List` reference variable but not vice versa. The reason is that `List` is an interface and interfaces can't be instantiated.

```
List<String> list6 = new ArrayList<>();
ArrayList<String> list7 = new List<>(); // DOES NOT COMPILE
```

Using an *ArrayList*

`ArrayList` has many methods, but you only need to know a handful of them—even fewer than you did for `String` and `StringBuilder`.

Before reading any further, you are going to see something new in the method signatures: a "class" named `E`. Don't worry—it isn't really a class. `E` is used by convention in generics to mean "any class that this array can hold." If you didn't specify a type when creating the `ArrayList`, `E` means `Object`. Otherwise, it means the class you put between `<` and `>`.

You should also know that `ArrayList` implements `toString()`, so you can easily see the contents just by printing it. Arrays do not produce such pretty output by default.

add()

The `add()` methods insert a new value in the `ArrayList`. The method signatures are as follows:

```
boolean add(E element)
void add(int index, E element)
```

Don't worry about the `boolean` return value. It always returns `true`. As we'll see later in the chapter, it is there because other classes in the `Collections` family need a return value in the signature when adding an element.

Since `add()` is the most critical `ArrayList` method you need to know for the exam, we are going to show a few examples for it. Let's start with the most straightforward case:

```
ArrayList list = new ArrayList();
list.add("hawk");          // [hawk]
list.add(Boolean.TRUE);    // [hawk, true]
System.out.println(list);  // [hawk, true]
```

`add()` does exactly what we expect: It stores the `String` in the no longer empty `ArrayList`. It then does the same thing for the `Boolean`. This is okay because we didn't specify a type for `ArrayList`; therefore, the type is `Object`, which includes everything except primitives. It may not have been what we intended, but the compiler doesn't know that. Now, let's use generics to tell the compiler we only want to allow `String` objects in our `ArrayList`:

```
ArrayList<String> safer = new ArrayList<>();
safer.add("sparrow");
safer.add(Boolean.TRUE);    // DOES NOT COMPILE
```

This time the compiler knows that only `String` objects are allowed in and prevents the attempt to add a `Boolean`. Now let's try adding multiple values to different positions.

```
4: List<String> birds = new ArrayList<>();
5: birds.add("hawk");            // [hawk]
6: birds.add(1, "robin");        // [hawk, robin]
7: birds.add(0, "blue jay");     // [blue jay, hawk, robin]
```

```
8: birds.add(1, "cardinal");     // [blue jay, cardinal, hawk, robin]
9: System.out.println(birds);    // [blue jay, cardinal, hawk, robin]
```

When a question has code that adds objects at indexed positions, draw it so that you won't lose track of which value is at which index. In this example, line 5 adds `"hawk"` to the end of `birds`. Then line 6 adds `"robin"` to index 1 of `birds`, which happens to be the end. Line 7 adds `"blue jay"` to index 0, which happens to be the beginning of `birds`. Finally, line 8 adds `"cardinal"` to index 1, which is now near the middle of `birds`.

remove()

The `remove()` methods remove the first matching value in the `ArrayList` or remove the element at a specified index. The method signatures are as follows:

```
boolean remove(Object object)
E remove(int index)
```

This time the `boolean` return value tells us whether a match was removed. The `E` return type is the element that actually got removed. The following shows how to use these methods:

```
3: List<String> birds = new ArrayList<>();
4: birds.add("hawk");     // [hawk]
5: birds.add("hawk");     // [hawk, hawk]
6: System.out.println(birds.remove("cardinal")); // prints false
7: System.out.println(birds.remove("hawk"));     // prints true
8: System.out.println(birds.remove(0));          // prints hawk
9: System.out.println(birds);                    // []
```

Line 6 tries to remove an element that is not in `birds`. It returns `false` because no such element is found. Line 7 tries to remove an element that is in `birds` and so returns `true`. Notice that it removes only one match.

Line 8 removes the element at index 0, which is the last remaining element in the `ArrayList`.

Since calling `remove()` with an `int` uses the index, an index that doesn't exist will throw an exception. For example, `birds.remove(100)` throws an `IndexOutOfBoundsException`.

There is also a `removeIf()` method. We'll cover it in the next chapter because it uses lambda expressions (a topic in that chapter).

set()

The `set()` method changes one of the elements of the `ArrayList` without changing the size. The method signature is as follows:

```
E set(int index, E newElement)
```

The `E` return type is the element that got replaced. The following shows how to use this method:

```
15: List<String> birds = new ArrayList<>();
16: birds.add("hawk");                    // [hawk]
17: System.out.println(birds.size());     // 1
18: birds.set(0, "robin");               // [robin]
19: System.out.println(birds.size());     // 1
20: birds.set(1, "robin");               // IndexOutOfBoundsException
```

Line 16 adds one element to the array, making the size 1. Line 18 replaces that one element, and the size stays at 1. Line 20 tries to replace an element that isn't in the `ArrayList`. Since the size is 1, the only valid index is 0. Java throws an exception because this isn't allowed.

isEmpty() and size()

The `isEmpty()` and `size()` methods look at how many of the slots are in use. The method signatures are as follows:

```
boolean isEmpty()
int size()
```

The following shows how to use these methods:

```
List<String> birds = new ArrayList<>();
System.out.println(birds.isEmpty());     // true
System.out.println(birds.size());        // 0
birds.add("hawk");                       // [hawk]
birds.add("hawk");                       // [hawk, hawk]
System.out.println(birds.isEmpty());     // false
System.out.println(birds.size());        // 2
```

At the beginning, `birds` has a size of 0 and is empty. It has a capacity that is greater than 0. However, as with `StringBuilder`, we don't use the capacity in determining size or length. After adding elements, the size becomes positive, and it is no longer empty. Notice how `isEmpty()` is a convenience method for `size() == 0`.

clear()

The `clear()` method provides an easy way to discard all elements of the `ArrayList`. The method signature is as follows:

```
void clear()
```

The following shows how to use this method:

```
List<String> birds = new ArrayList<>();
birds.add("hawk");                    // [hawk]
birds.add("hawk");                    // [hawk, hawk]
System.out.println(birds.isEmpty());  // false
```

```
System.out.println(birds.size());     // 2
birds.clear();                        // []
System.out.println(birds.isEmpty());  // true
System.out.println(birds.size());     // 0
```

After we call `clear()`, `birds` is back to being an empty `ArrayList` of size 0.

contains()

The `contains()` method checks whether a certain value is in the `ArrayList`. The method signature is as follows:

```
boolean contains(Object object)
```

The following shows how to use this method:

```
List<String> birds = new ArrayList<>();
birds.add("hawk");                            // [hawk]
System.out.println(birds.contains("hawk"));  // true
System.out.println(birds.contains("robin")); // false
```

This method calls `equals()` on each element of the `ArrayList` to see whether there are any matches. Since `String` implements `equals()`, this works out well.

equals()

Finally, `ArrayList` has a custom implementation of `equals()`,so you can compare two lists to see whether they contain the same elements in the same order.

```
boolean equals(Object object)
```

The following shows an example:

```
31: List<String> one = new ArrayList<>();
32: List<String> two = new ArrayList<>();
33: System.out.println(one.equals(two));  // true
34: one.add("a");                         // [a]
35: System.out.println(one.equals(two));  // false
36: two.add("a");                         // [a]
37: System.out.println(one.equals(two));  // true
38: one.add("b");                         // [a,b]
39: two.add(0, "b");                      // [b,a]
40: System.out.println(one.equals(two));  // false
```

On line 33, the two `ArrayList` objects are equal. An empty list is certainly the same elements in the same order. On line 35, the `ArrayList` objects are not equal because the size is different. On line 37, they are equal again because the same one element is in each. On line 40, they are not equal. The size is the same and the values are the same, but they are not in the same order.

Wrapper Classes

Up to now, we've only put `String` objects in the `ArrayList`. What happens if we want to put primitives in? Each primitive type has a wrapper class, which is an object type that corresponds to the primitive. Table 5.4 lists all the wrapper classes along with how to create them.

TABLE 5.4 Wrapper classes

Primitive type	Wrapper class	Example of creating
`boolean`	`Boolean`	`Boolean.valueOf(true)`
`byte`	`Byte`	`Byte.valueOf((byte) 1)`
`short`	`Short`	`Short.valueOf((short) 1)`
`int`	`Integer`	`Integer.valueOf(1)`
`long`	`Long`	`Long.valueOf(1)`
`float`	`Float`	`Float.valueOf((float) 1.0)`
`double`	`Double`	`Double.valueOf(1.0)`
`char`	`Character`	`Character.valueOf('c')`

Each wrapper class also has a constructor. It works the same way as `valueOf()` but isn't recommended for new code. The `valueOf()` allows object caching. Remember how a `String` could be shared when the value is the same? The wrapper classes are immutable and take advantage of some caching as well.

The wrapper classes also have a method that converts back to a primitive. You don't need to know much about the `valueOf()` or i`ntValue()` type methods for the exam because autoboxing has removed the need for them (see the next section). You just need to be able to read the code and not look for tricks in it.

There are also methods for converting a `String` to a primitive or wrapper class. You do need to know these methods. The parse methods, such

as `parseInt()`, return a primitive, and the `valueOf()` method returns a wrapper class. This is easy to remember because the name of the returned primitive is in the method name. Here's an example:

```
int primitive = Integer.parseInt("123");
Integer wrapper = Integer.valueOf("123");
```

The first line converts a `String` to an `int` primitive. The second converts a `String` to an `Integer` wrapper class. If the `String` passed in is not valid for the given type, Java throws an exception. In these examples, letters and dots are not valid for an integer value:

```
int bad1 = Integer.parseInt("a");         // throws NumberFormatException
Integer bad2 = Integer.valueOf("123.45"); // throws NumberFormatException
```

Before you worry, the exam won't make you recognize that the method `parseInt()` is used rather than `parseInteger()`. You simply need to be able to recognize the methods when put in front of you. Also, the `Character` class doesn't participate in the `parse/valueOf` methods. Since a `String` consists of characters, you can just call `charAt()` normally.

Table 5.5 lists the methods you need to recognize for creating a primitive or wrapper class object from a `String`. In real coding, you won't be so concerned about which is returned from each method due to autoboxing.

TABLE 5.5 Converting from a `String`

Wrapper class	Converting `String` to a primitive	Converting `String` to a wrapper class
`Boolean`	`Boolean.parseBoolean("true")`	`Boolean.valueOf("TRUE")`
`Byte`	`Byte.parseByte("1")`	`Byte.valueOf("2")`
`Short`	`Short.parseShort("1")`	`Short.valueOf("2")`
`Integer`	`Integer.parseInt("1")`	`Integer.valueOf("2")`
`Long`	`Long.parseLong("1")`	`Long.valueOf("2")`
`Float`	`Float.parseFloat("1")`	`Float.valueOf("2.2")`
`Double`	`Double.parseDouble("1")`	`Double.valueOf("2.2")`
`Character`	None	None

WRAPPER CLASSES AND NULL

When we presented numeric primitives in Chapter 2, we mentioned they could not be used to store `null` values. One advantage of a wrapper class over a primitive is that because it's an object, it can be used to store a `null` value. While `null` values aren't particularly useful for numeric calculations, they are quite useful in data-based services. For example, if you are storing a user's location data using (latitude,longitude), it would be a bad idea to store a missing point as (0,0) since that refers to an actual location off the coast of Africa where the user could theoretically be.

Autoboxing and Unboxing

Why won't you need to be concerned with whether a primitive or wrapper class is returned, you ask? Since Java 5, you can just type the primitive value, and Java will convert it to the relevant wrapper class for you. This is called *autoboxing*. The reverse conversion of wrapper class to primitive value is called *unboxing*. Let's look at an example:

```
3: List<Integer> weights = new ArrayList<>();
4: Integer w = 50;
5: weights.add(w);                  // [50]
6: weights.add(Integer.valueOf(60));     // [50, 60]
7: weights.remove(new Integer(50)); // [60]
8: double first = weights.get(0);   // 60.0
```

Line 4 autoboxes the `int` primitive into an `Integer` object, and line 5 adds that to the `List`. Line 6 shows that you can still write code the long way and pass in a wrapper object. Line 8 retrieves the first `Integer` in the list, unboxes it as a primitive and implicitly casts it to `double`.

What do you think happens if you try to unbox a `null`?

```
3: List<Integer> heights = new ArrayList<>();
4: heights.add(null);
5: int h = heights.get(0);          // NullPointerException
```

On line 4, we add a `null` to the list. This is legal because a `null` reference can be assigned to any reference variable. On line 5, we try to unbox that `null` to an `int` primitive. This is a problem. Java tries to get the `int` value of `null`. Since calling any method on `null` gives a `NullPointerException`, that is just what we get. Be careful when you see `null` in relation to autoboxing.

Also be careful when autoboxing into `Integer`. What do you think this code outputs?

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.remove(1);
System.out.println(numbers);
```

It actually outputs [1]. After adding the two values, the `List` contains [1, 2]. We then request the element with index 1 be removed. That's right: index 1. Because there's already a `remove()` method that takes an `int` parameter, Java calls that method rather than autoboxing. If you want to remove the 1, you can write `numbers.remove(new Integer(1))` to force wrapper class use.

Converting Between *array* and *List*

You should know how to convert between an array and a `List`. Let's start with turning an `ArrayList` into an array:

```
13: List<String> list = new ArrayList<>();
14: list.add("hawk");
15: list.add("robin");
16: Object[] objectArray = list.toArray();
17: String[] stringArray = list.toArray(new String[0]);
18: list.clear();
19: System.out.println(objectArray.length);     // 2
20: System.out.println(stringArray.length);     // 2
```

Line 16 shows that an `ArrayList` knows how to convert itself to an array. The only problem is that it defaults to an array of class `Object`. This isn't usually what you want. Line 17 specifies the type of the array and does what we actually want. The advantage of specifying a size of 0 for the parameter is that Java will create a new array of the proper size for the return value. If you like, you can suggest a larger array to be used instead. If the `ArrayList` fits in that array, it will be returned. Otherwise, a new one will be created.

Also, notice that line 18 clears the original `List`. This does not affect either array. The array is a newly created object with no relationship to the original `List`. It is simply a copy.

Converting from an array to a `List` is more interesting. We will show you two methods to do this conversion. Note that you aren't guaranteed to get a `java.util.ArrayList` from either. This means each has special behavior to learn about.

One option is to create a `List` that is linked to the original array. When a change is made to one, it is available in the other. It is a fixed-size list and is also known as a backed `List` because the array changes with it. Pay careful attention to the values here:

```
20: String[] array = { "hawk", "robin" };     // [hawk, robin]
21: List<String> list = Arrays.asList(array); // returns fixed size list
22: System.out.println(list.size());         // 2
23: list.set(1, "test");                      // [hawk, test]
24: array[0] = "new";                         // [new, test]
25: System.out.print(Arrays.toString(array));// [new, test]
26: list.remove(1);     // throws UnsupportedOperationException
```

Line 21 converts the array to a `List`. Note that it isn't the `java.util.ArrayList` we've grown used to. It is a fixed-size, backed version of a `List`. Line 23 is okay because `set()` merely replaces an existing value. It updates both `array` and `list` because they point to the same data store. Line 24 also changes both `array` and `list`. Line 25 shows the array has changed to `[new, test]`. Line 26 throws an exception because we are not allowed to change the size of the list.

Another option is to create an immutable `List`. That means you cannot change the values or the size of the `List`. You can change the original array, but changes will not be reflected in the immutable `List`. Again, pay careful attention to the values:

```
32: String[] array = { "hawk", "robin" };        // [hawk, robin]
33: List<String> list = List.of(array);      // returns immutable list
34: System.out.println(list.size());            // 2
35: array[0] = "new";
36: System.out.println(Arrays.toString(array)); // [new, robin]
37: System.out.println(list);                   // [hawk, robin]
38: list.set(1, "test");      // throws UnsupportedOperationException
```

Line 33 creates the immutable `List`. It contains the two values that `array` happened to contain at the time the `List` was created. On line 35, there is a change to the array. Line 36 shows that `array` has changed. Line 37 shows that `list` still has the original values. This is because it is an immutable copy of the original array. Line 38 shows that changing a list value in an immutable list is not allowed.

Using Varargs to Create a List

Using varargs allows you to create a `List` in a cool way:

```
List<String> list1 = Arrays.asList("one", "two");
List<String> list2 = List.of("one", "two");
```

Both of these methods take varargs, which let you pass in an array or just type out the `String` values. This is handy when testing because you can easily create and populate a `List` on one line. Both methods create fixed-size arrays. If you will need to later add or remove elements, you'll still need to create an `ArrayList` using the constructor. There's a lot going on here, so let's study Table 5.6.

TABLE 5.6 Array and list conversions

	`toArray()`	`Arrays.asList()`	`List.of()`
Type converting from	`List`	Array (or varargs)	Array (or varargs)
Type created	Array	`List`	`List`
Allowed to remove values from created object	No	No	No
Allowed to change values in the created object	Yes	Yes	No
Changing values in the created object affects the original or vice versa.	No	Yes	N/A

Notice that none of the options allows you to change the number of elements. If you want to do that, you'll need to actually write logic to create the new object. Here's an example:

```
List<String> fixedSizeList  = Arrays.asList("a", "b", "c");
List<String> expandableList = new ArrayList<>(fixedSizeList);
```

Sorting

Sorting an `ArrayList` is similar to sorting an array. You just use a different helper class:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(99);
numbers.add(5);
numbers.add(81);
Collections.sort(numbers);
System.out.println(numbers); // [5, 81, 99]
```

As you can see, the numbers got sorted, just like you'd expect. Isn't it nice to have something that works just like you think it will?

Creating Sets and Maps

Although advanced collections topics are not covered until the 1Z0-816 exam, you should still know the basics of `Set` and `Map` now.

Introducing Sets

A `Set` is a collection of objects that cannot contain duplicates. If you try to add a duplicate to a set, the API will not fulfill the request. You can imagine a set as shown in Figure 5.9.

FIGURE 5.9 **Example of a `Set`**

All the methods you learned for `ArrayList` apply to a `Set` with the exception of those taking an index as a parameter. Why is this? Well, a `Set` isn't ordered, so it wouldn't make sense to talk about the first element. This means you cannot call `set(index, value)` or `remove(index)`. You can call other methods like `add(value)` or `remove(value)`.

Do you remember that `boolean` return value on `add()` that always returned `true` for an `ArrayList`? `Set` is a reason it needs to exist. When trying to add a duplicate value, the method returns `false` and does not add the value.

There are two common classes that implement `Set` that you might see on the exam. `HashSet` is the most common. `TreeSet` is used when sorting is important.

To make sure you understand a `Set`, follow along with this code:

```
3: Set<Integer> set = new HashSet<>();
4: System.out.println(set.add(66)); // true
```

```
5: System.out.println(set.add(66)); // false
6: System.out.println(set.size()); // 1
7: set.remove(66);
8: System.out.println(set.isEmpty()); // true
```

Line 3 creates a new set that declares only unique elements are allowed. Both lines 4 and 5 attempt to add the same value. Only the first one is allowed, making line 4 print `true` and line 5 `false`. Line 6 confirms there is only one value in the set. Removing an element on line 7 works normally, and the set is empty on line 8.

Introducing Maps

A `Map` uses a key to identify values. For example, when you use the contact list on your phone, you look up "George" rather than looking through each phone number in turn. Figure 5.10 shows how to visualize a `Map`.

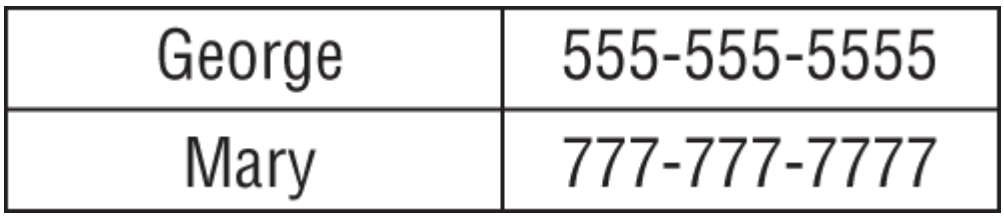

FIGURE 5.10 Example of a `Map`

The most common implementation of `Map` is `HashMap`. Some of the methods are the same as those in `ArrayList` like `clear()`, `isEmpty()`, and `size()`.

There are also methods specific to dealing with key and value pairs. Table 5.7 shows these minimal methods you need to know.

TABLE 5.7 Common `Map` methods

Method	Description
`V get(Object key)`	Returns the value mapped by key or `null` if none is mapped
`V getOrDefault(Object key, V other)`	Returns the value mapped by key or `other` if none is mapped
`V put(K key, V value)`	Adds or replaces key/value pair. Returns previous value or `null`
`V remove(Object key)`	Removes and returns value mapped to key. Returns `null` if none
`boolean containsKey(Object key)`	Returns whether key is in map
`boolean containsValue(Object value)`	Returns whether value is in map
`Set<K> keySet()`	Returns set of all keys
`Collection<V> values()`	Returns `Collection` of all values

Now let's look at an example to confirm this is clear:

```
8:  Map<String, String> map = new HashMap<>();
9:  map.put("koala", "bamboo");
10: String food = map.get("koala"); // bamboo
```

```
11: String other = map.getOrDefault("ant", "leaf"); // leaf
12: for (String key: map.keySet())
13:    System.out.println(key + " " + map.get(key)); // koala bamboo
```

In this example, we create a new map and store one key/value pair inside. Line 10 gets this value by key. Line 11 looks for a key that isn't there, so it returns the second parameter `leaf` as the default value. Lines 12 and 13 list all the key and value pairs.

Calculating with Math APIs

It should come as no surprise that computers are good at computing numbers. Java comes with a powerful `Math` class with many methods to make your life easier. We will just cover a few common ones here that are most likely to appear on the exam. When doing your own projects, look at the `Math` Javadoc to see what other methods can help you.

Pay special attention to return types in math questions. They are an excellent opportunity for trickery!

min() and *max()*

The `min()` and `max()` methods compare two values and return one of them.

The method signatures for `min()` are as follows:

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

There are four overloaded methods, so you always have an API available with the same type. Each method returns whichever of `a` or `b` is smaller.

The `max()` method works the same way except it returns the larger value.

The following shows how to use these methods:

```
int first = Math.max(3, 7);  // 7
int second = Math.min(7, -9);  // -9
```

The first line returns 7 because it is larger. The second line returns -9 because it is smaller. Remember from school that negative values are smaller than positive ones.

round()

The `round()` method gets rid of the decimal portion of the value, choosing the next higher number if appropriate. If the fractional part is .5 or higher, we round up.

The method signatures for `round()` are as follows:

```
long round(double num)
int round(float num)
```

There are two overloaded methods to ensure there is enough room to store a rounded `double` if needed. The following shows how to use this method:

```
long low = Math.round(123.45); // 123
long high = Math.round(123.50); // 124
int fromFloat = Math.round(123.45f); // 123
```

The first line returns 123 because .45 is smaller than a half. The second line returns 124 because the fractional part is just barely a half. The final

line shows that an explicit float triggers the method signature that returns an `int`.

pow()

The `pow()` method handles exponents. As you may recall from your elementary school math class, 3^2 means three squared. This is 3 * 3 or 9. Fractional exponents are allowed as well. Sixteen to the .5 power means the square root of 16, which is 4. (Don't worry, you won't have to do square roots on the exam.)

The method signature is as follows:

```
double pow(double number, double exponent)
```

The following shows how to use this method:

```
double squared = Math.pow(5, 2); // 25.0
```

Notice that the result is 25.0 rather than 25 since it is a `double`. (Again, don't worry, the exam won't ask you to do any complicated math.)

random()

The `random()` method returns a value greater than or equal to 0 and less than 1. The method signature is as follows:

```
double random()
```

The following shows how to use this method:

```
double num = Math.random();
```

Since it is a random number, we can't know the result in advance. However, we can rule out certain numbers. For example, it can't be negative because that's less than 0. It can't be 1.0 because that's not less than 1.

Summary

In this chapter, you learned that `String`s are immutable sequences of characters. The `new` operator is optional. The concatenation operator (`+`) creates a new `String` with the content of the first `String` followed by the content of the second `String`. If either operand involved in the `+` expression is a `String`, concatenation is used; otherwise, addition is used. `String` literals are stored in the string pool. The `String` class has many methods.

`StringBuilder`s are mutable sequences of characters. Most of the methods return a reference to the current object to allow method chaining. The `StringBuilder` class has many methods.

Calling `==` on `String` objects will check whether they point to the same object in the pool. Calling `==` on `StringBuilder` references will check whether they are pointing to the same `StringBuilder` object. Calling `equals()` on `String` objects will check whether the sequence of characters is the same. Calling `equals()` on `StringBuilder` objects will check whether they are pointing to the same object rather than looking at the values inside.

An array is a fixed-size area of memory on the heap that has space for primitives or pointers to objects. You specify the size when creating it—for example, `int[] a = new int[6];`. Indexes begin with 0, and elements are referred to using `a[0]`. The `Arrays.sort()` method sorts an array. `Arrays.binarySearch()` searches a sorted array and returns the index of a match. If no match is found, it negates the position where the element would need to be inserted and subtracts 1. `Arrays.compare()` and `Arrays .mismatch()` check whether two arrays are the equivalent. Methods that are passed varargs (...) can be used as if a normal array was

passed in. In a multidimensional array, the second-level arrays and beyond can be different sizes.

An `ArrayList` can change size over its life. It can be stored in an `ArrayList` or `List` reference. Generics can specify the type that goes in the `ArrayList`. Although an `ArrayList` is not allowed to contain primitives, Java will autobox parameters passed in to the proper wrapper type. `Collections.sort()` sorts an `ArrayList`.

A `Set` is a collection with unique values. A `Map` consists of key/value pairs. The `Math` class provides many static methods to facilitate programming.

Exam Essentials

Be able to determine the output of code using `String`. Know the rules for concatenating `String`s and how to use common `String` methods. Know that `String`s are immutable. Pay special attention to the fact that indexes are zero-based and that `substring()` gets the string up until right before the index of the second parameter.

Be able to determine the output of code using `StringBuilder`. Know that `StringBuilder` is mutable and how to use common `StringBuilder` methods. Know that `substring()` does not change the value of a `StringBuilder`, whereas `append()`, `delete()`, and `insert()` do change it. Also note that most `StringBuilder` methods return a reference to the current instance of `StringBuilder`.

Understand the difference between `==` and `equals()`. `==` checks object equality. `equals()` depends on the implementation of the object it is being called on. For `String`s, `equals()` checks the characters inside of it.

Be able to determine the output of code using arrays. Know how to declare and instantiate one-dimensional and multidimensional arrays. Be

able to access each element and know when an index is out of bounds. Recognize correct and incorrect output when searching and sorting.

Be able to determine the output of code using `ArrayList`. Know that `ArrayList` can increase in size. Be able to identify the different ways of declaring and instantiating an `ArrayList`. Identify correct output from `ArrayList` methods, including the impact of autoboxing.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. What is output by the following code? (Choose all that apply.)

```
1: public class Fish {
2:    public static void main(String[] args) {
3:       int numFish = 4;
4:       String fishType = "tuna";
5:       String anotherFish = numFish + 1;
6:       System.out.println(anotherFish + " " + fishType);
7:       System.out.println(numFish + " " + 1);
8: } }
```

   a. `4 1`
   b. `5`
   c. `5 tuna`
   d. `5tuna`
   e. `51tuna`
   f. The code does not compile.

2. Which of the following are output by this code? (Choose all that apply.)

```
3: var s = "Hello";
4: var t = new String(s);
5: if ("Hello".equals(s)) System.out.println("one");
6: if (t == s) System.out.println("two");
```

```
7: if (t.intern() == s) System.out.println("three");
8: if ("Hello" == s) System.out.println("four");
9: if ("Hello".intern() == t) System.out.println("five");
```

1. `one`
2. `two`
3. `three`
4. `four`
5. `five`
6. The code does not compile.
7. None of the above

3. Which statements about the following code snippet are correct? (Choose all that apply.)

```
List<String> gorillas = new ArrayList<>();
for(var koko : gorillas)
   System.out.println(koko);

var monkeys = new ArrayList<>();
for(var albert : monkeys)
   System.out.println(albert);

List chimpanzees = new ArrayList<Integer>();
for(var ham : chimpanzees)
   System.out.println(ham);
```

1. The data type of `koko` is `String`.
2. The data type of `koko` is `Object`.
3. The data type of `albert` is `Object`.
4. The data type of `albert` is undefined.
5. The data type of `ham` is `Integer`.
6. The data type of `ham` is `Object`.
7. None of the above, as the code does not compile

4. What is the result of the following code?

```
7: StringBuilder sb = new StringBuilder();
8: sb.append("aaa").insert(1, "bb").insert(4, "ccc");
9: System.out.println(sb);
```

1. `abbaaccc`
2. `abbaccca`
3. `bbaaaccc`
4. `bbaaccca`
5. An empty line
6. The code does not compile.

5. What is the result of the following code?

```
12: int count = 0;
13: String s1 = "java";
14: String s2 = "java";
15: StringBuilder s3 = new StringBuilder("java");
16: if (s1 == s2) count++;
17: if (s1.equals(s2)) count++;
18: if (s1 == s3) count++;
19: if (s1.equals(s3)) count++;
20: System.out.println(count);
```

1. `0`
2. `1`
3. `2`
4. `3`
5. `4`
6. An exception is thrown.
7. The code does not compile.

6. What is the result of the following code?

```
public class Lion {
   public void roar(String roar1, StringBuilder roar2) {
      roar1.concat("!!!");
      roar2.append("!!!");
```

```
    }
    public static void main(String[] args) {
       String roar1 = "roar";
       StringBuilder roar2 = new StringBuilder("roar");
       new Lion().roar(roar1, roar2);
       System.out.println(roar1 + " " + roar2);
} }
```

1. `roar roar`
2. `roar roar!!!`
3. `roar!!! roar`
4. `roar!!! roar!!!`
5. An exception is thrown.
6. The code does not compile.

7. Which of the following return the number `5` when run independently? (Choose all that apply.)

```
var string = "12345";
var builder = new StringBuilder("12345");
```

1. `builder.charAt(4)`
2. `builder.replace(2, 4, "6").charAt(3)`
3. `builder.replace(2, 5, "6").charAt(2)`
4. `string.charAt(5)`
5. `string.length`
6. `string.replace("123", "1").charAt(2)`
7. None of the above

8. What is output by the following code? (Choose all that apply.)

```
String numbers = "012345678";
System.out.println(numbers.substring(1, 3));
System.out.println(numbers.substring(7, 7));
System.out.println(numbers.substring(7));
```

1. `12`

2. `123`
3. `7`
4. `78`
5. A blank line
6. The code does not compile.
7. An exception is thrown.

9. What is the result of the following code? (Choose all that apply.)

```
style
14: String s1 = "purr";
15: String s2 = "";
16:
17: s1.toUpperCase();
18: s1.trim();
19: s1.substring(1, 3);
20: s1 += "two";
21:
22: s2 += 2;
23: s2 += 'c';
24: s2 += false;
25:
26: if ( s2 == "2cfalse") System.out.println("==");
27: if ( s2.equals("2cfalse")) System.out.println("equals");
28: System.out.println(s1.length());
```

1. `2`
2. `4`
3. `7`
4. `10`
5. `==`
6. `equals`
7. An exception is thrown.
8. The code does not compile.

10. Which of these statements are true? (Choose all that apply.)

```
var letters = new StringBuilder("abcdefg");
```

1. `letters.substring(1, 2)` returns a single character `String`.
2. `letters.substring(2, 2)` returns a single character `String`.
3. `letters.substring(6, 5)` returns a single character `String`.
4. `letters.substring(6, 6)` returns a single character `String`.
5. `letters.substring(1, 2)` throws an exception.
6. `letters.substring(2, 2)` throws an exception.
7. `letters.substring(6, 5)` throws an exception.
8. `letters.substring(6, 6)` throws an exception.

11. What is the result of the following code?

```
StringBuilder numbers = new StringBuilder("0123456789");
numbers.delete(2,  8);
numbers.append("-").insert(2, "+");
System.out.println(numbers);
```

1. `01+89-`
2. `012+9-`
3. `012+-9`
4. `0123456789`
5. An exception is thrown.
6. The code does not compile.

12. What is the result of the following code?

```
StringBuilder b = "rumble";
b.append(4).deleteCharAt(3).delete(3, b.length() - 1);
System.out.println(b);
```

1. `rum`
2. `rum4`
3. `rumb4`
4. `rumble4`
5. An exception is thrown.
6. The code does not compile.

13. Which of the following can replace line 4 to print `"avaJ"` ? (Choose all that apply.)

```
3: var puzzle = new StringBuilder("Java");
4: // INSERT CODE HERE
5: System.out.println(puzzle);
```

1. `puzzle.reverse();`
2. `puzzle.append("vaJ$").substring(0, 4);`
3. `puzzle.append("vaJ$").delete(0, 3).deleteCharAt(puzzle.length() - 1);`
4. `puzzle.append("vaJ$").delete(0, 3).deleteCharAt(puzzle.length());`
5. None of the above

14. Which of these array declarations is not legal? (Choose all that apply.)
    1. `int[][] scores = new int[5][];`
    2. `Object[][][] cubbies = new Object[3][0][5];`
    3. `String beans[] = new beans[6];`
    4. `java.util.Date[] dates[] = new java.util.Date[2][];`
    5. `int[][] types = new int[];`
    6. `int[][] java = new int[][];`
15. Which of the following can fill in the blanks so the code compiles? (Choose two.)

```
6: char[]c = new char[2];
7: ArrayList l = new ArrayList();
8: int length = __________ + _______________;
```

1. `c.length`
2. `c.length()`
3. `c.size`
4. `c.size()`
5. `l.length`
6. `l.length()`

7. `l.size`
8. `l.size()`

16. Which of the following are true? (Choose all that apply.)
    1. An array has a fixed size.
    2. An `ArrayList` has a fixed size.
    3. An array is immutable.
    4. An `ArrayList` is immutable.
    5. Calling `equals()` on two equivalent arrays returns `true`.
    6. Calling `equals()` on two equivalent `ArrayList` objects returns `true`.
    7. If you call `remove(0)` using an empty `ArrayList` object, it will compile successfully.
    8. If you call `remove(0)` using an empty `ArrayList` object, it will run successfully.
17. What is the result of the following statements?

```
6:  var list = new ArrayList<String>();
7:  list.add("one");
8:  list.add("two");
9:  list.add(7);
10: for(var s : list)  System.out.print(s);
```

    1. `onetwo`
    2. `onetwo7`
    3. `onetwo` followed by an exception
    4. Compiler error on line 6
    5. Compiler error on line 7
    6. Compiler error on line 9
    7. Compiler error on line 10
18. Which of the following pairs fill in the blanks to output `6`?

```
3: var values = new _____________<Integer>();
4: values.add(4);
5: values.add(4);
6: values._____________;
```

```
7: values.remove(0);
8: for (var v : values) System.out.print(v);
```

1. `ArrayList` and `put(1, 6)`
2. `ArrayList` and `replace(1, 6)`
3. `ArrayList` and `set(1, 6)`
4. `HashSet` and `put(1, 6)`
5. `HashSet` and `replace(1, 6)`
6. `HashSet` and `set(1, 6)`
7. The code does not compile with any of these options.

19. What is output by the following? (Choose all that apply.)

```
8:  List<Integer> list = Arrays.asList(10, 4, -1, 5);
9:  int[] array = { 6, -4, 12, 0, -10 };
10: Collections.sort(list);
11:
12: Integer converted[] = list.toArray(new Integer[4]);
13: System.out.println(converted[0]);
14: System.out.println(Arrays.binarySearch(array, 12));
```

1. `-1`
2. `2`
3. `4`
4. `6`
5. `10`
6. One of the outputs is undefined.
7. An exception is thrown.
8. The code does not compile.

20. Which of the lines contain a compiler error? (Choose all that apply.)

```
23: double one = Math.pow(1, 2);
24: int two = Math.round(1.0);
25: float three = Math.random();
26: var doubles = new double[] { one, two, three};
27:
28: String [] names = {"Tom", "Dick", "Harry"};
```

```
29: List<String> list = names.asList();
30: var other = Arrays.asList(names);
31: other.set(0, "Sue");
```

1. Line 23
2. Line 24
3. Line 25
4. Line 26
5. Line 29
6. Line 30
7. Line 31

21. What is the result of the following?

```
List<String> hex = Arrays.asList("30", "8", "3A", "FF");
Collections.sort(hex);
int x = Collections.binarySearch(hex, "8");
int y = Collections.binarySearch(hex, "3A");
int z = Collections.binarySearch(hex, "4F");
System.out.println(x + " " + y + " " + z);
```

1. `0 1 -2`
2. `0 1 -3`
3. `2 1 -2`
4. `2 1 -3`
5. None of the above
6. The code doesn't compile.

22. Which of the following are true statements about the following code? (Choose all that apply.)

```
4: List<Integer> ages = new ArrayList<>();
5: ages.add(Integer.parseInt("5"));
6: ages.add(Integer.valueOf("6"));
7: ages.add(7);
8: ages.add(null);
9: for (int age : ages) System.out.print(age);
```

1. The code compiles.
2. The code throws a runtime exception.
3. Exactly one of the add statements uses autoboxing.
4. Exactly two of the add statements use autoboxing.
5. Exactly three of the add statements use autoboxing.

23. What is the result of the following?

```
List<String> one = new ArrayList<String>();
one.add("abc");
List<String> two = new ArrayList<>();
two.add("abc");
if (one == two)
   System.out.println("A");
else if (one.equals(two))
   System.out.println("B");
else
   System.out.println("C");
```

1. `A`
2. `B`
3. `C`
4. An exception is thrown.
5. The code does not compile.

24. Which statements are true about the following code? (Choose all that apply.)

```
public void run(Integer[] ints, Double[] doubles) {
   List<Integer> intList = Arrays.asList(ints);
   List<Double> doubleList = List.of(doubles);
   // more code
}
```

1. Adding an element to `doubleList` is allowed.
2. Adding an element to `intList` is allowed.
3. Changing the first element in `doubleList` changes the first element in `doubles`.

4. Changing the first element in `intList` changes the first element in `ints`.
5. `doubleList` is immutable.
6. `intList` is immutable.

25. Which of the following statements are true of the following code? (Choose all that apply.)

```
String[] s1 = { "Camel", "Peacock", "Llama"};
String[] s2 = { "Camel", "Llama", "Peacock"};
String[] s3 = { "Camel"};
String[] s4 = { "Camel", null};
```

1. `Arrays.compare(s1, s2)` returns a positive integer.
2. `Arrays.mismatch(s1, s2)` returns a positive integer.
3. `Arrays.compare(s3, s4)` returns a positive integer.
4. `Arrays.mismatch(s3, s4)` returns a positive integer.
5. `Arrays.compare(s4, s4)` returns a positive integer.
6. `Arrays.mismatch(s4, s4)` returns a positive integer.