The OCA exam expects you to know the core data structures and classes used in Java, and in this chapter readers will learn about the most common methods available. For example, `String` and `StringBuilder` are used for text data. An array and an `ArrayList` are used when you have multiple values. A variety of classes are used for working with dates. In this chapter, you'll also learn how to determine whether two objects are equal.

API stands for application programming interface. In Java, an interface is something special. In the context of an API, it can be a group of class or interface definitions that gives you access to a service or functionality. You will learn about the most common APIs for each of the classes covered in this chapter.

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 1, "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 of type *name* pointing to the `String` object `"Fluffy"`. They are subtly different, as you'll see in the section "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`.

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

Concatenation

In Chapter 2, "Operators and Statements," 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 together is called string *concatenation*. The OCA 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
```

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? 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 OCA 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 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` only has a getter. There's no way to change the value of *s* once it's set. `Mutable` has a setter as well. 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, and subclasses can't add 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 at you.

The String Pool

Since strings are everywhere in Java, they use up a lot of memory. In some production applications, they can use up 25–40 percent of the 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 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. Strings not in the string pool are garbage collected just like any other object.

Remember back when we said these two lines are subtly different?

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

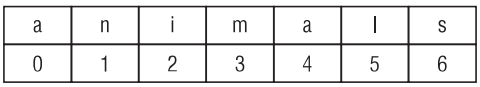
The former says to use the string pool normally. The second says "No, JVM. I really don't want you to use the string pool. Please create a new object for me even though it is less efficient." When you write programs, you wouldn't want to do this. For the exam, you need to know that it is allowed.

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 3.1 shows how each character in the string `"animals"` is indexed.

FIGURE 3.1 Indexing for a string



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

Let's look at thirteen 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.

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 6, "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. The method signatures are as follows:

```
int indexOf(char ch)
int indexOf(char ch, index fromIndex)
int indexOf(String str)
int indexOf(String str, index 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 and so matches later on. 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 returned 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:

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

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: 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 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 str)
String toUpperCase(String str)
```

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(String str)
boolean equalsIgnoreCase(String str)
```

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.

contains()

The `contains()` method also 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(String str)
```

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`**.

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. A `CharSequence` is a general way of representing several classes, including `String` and `StringBuilder`. It's called an interface, which we'll cover in Chapter 5, "Class Design." The method signatures are as follows:

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

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.

trim()

You've made it through the all the `String` methods you need to know except one. We left the easy one for last. The `trim()` method removes 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 method signature is as follows:

```
public String trim()
```

The following code shows how to use this method:

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

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.

Method Chaining

It is common to call multiple methods on the same `String`, 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 or 7 changes *a*.

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 and appends the value of current to the end of *alpha*. This code reuses the same `StringBuilder` without creating an interim `String` each time.

In this section, we'll look at creating a `StringBuilder`, common methods, and a comparison to `StringBuffer`.

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` objects 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 exact same object and would print out the same value.

The exam won't always make the code easy to read by only having 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 number of slots for characters.

Size vs. Capacity

The behind-the-scenes process of how objects are stored isn't on the exam, but some knowledge of this process may help you better understand and remember `StringBuilder`.

Size is the number of characters currently in the sequence, and capacity is the number of characters the sequence can currently hold. Since a `String` is immutable, the size and capacity are the same. The number of characters appearing in the `String` is both the size and capacity.

For `StringBuilder`, Java knows the size is likely to change as the object is used. When `StringBuilder` is constructed, it may start at the default capacity (which happens to be 16) or one of the programmer's choosing. In the example, we request a capacity of 5. At this point, the size is 0 since no characters have been added yet, but we have space for 5.

continues

(continued)

Next we add four characters. At this point, the size is 4 since four slots are taken. The capacity is still 5. Then we add three more characters. The size is now 7 since we have used up seven slots. Because the capacity wasn't large enough to store seven characters, Java automatically increased it for us.

```
StringBuilder sb = new StringBuilder(5);
```

0	1	2	3	4

```
sb.append("anim");
```

a	n	i	m	
0	1	2	3	4

```
sb.append("als");
```

a	n	i	m	a	l	s		
0	1	2	3	4	5	6	7	...

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 where the substring 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 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 start, int end)
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`.

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:

```
String s = sb.toString();
```