Chapter 4
Core APIs

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **Handling Date, Time, Text, Numeric and Boolean Values**
    - Use primitives and wrapper classes. Evaluate arithmetic and boolean expressions, using the Math API and by applying precedence rules, type conversions, and casting.
    - Manipulate text, including text blocks, using String and StringBuilder classes.
    - Manipulate date, time, duration, period, instant and time-zone objects including daylight saving time using Date-Time API.
- ✓ **Working with Arrays and Collections**
    - Create arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements.

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 functionality.

In this chapter, you 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. Then we cover arrays. Finally, we explore math and date/time APIs.

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, "Building Blocks," this is an example of a reference type. You also learned that objects 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 see later in this chapter. For now, just remember that the `String` class is special and doesn't need to be instantiated with `new`.

Further, text blocks are another way of creating a `String`. To review, this text block is the same as the previous variables:

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

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 learn more about interfaces in Chapter 7, "Beyond Classes."

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

Concatenating

In Chapter 2, "Operators," you learned how to add numbers. 1 + 2 is clearly 3. But what is `"1" + "2"`? It's `"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 lots 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
System.out.println("c" + null);      // cnull
```

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 used only in code; they don’t get output.

The third example combines 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 are used in one line?

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"`.

Finally, the last example shows how `null` is represented as a string when concatenated or printed, giving us `"cnull"`.

The exam takes trickery a step further and will try to fool 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, 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`. Then, 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 one more thing to know about concatenation, but it is easy. In this example, you just have to remember what `+=` does. Keep in mind, `s += "2"` means the same thing as `s = s + "2"`.

```
4: var 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
```

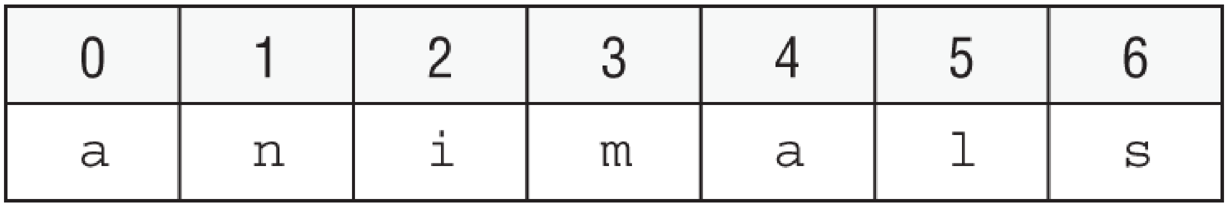
On line 5, we are “adding” two strings, which means we concatenate them. Line 6 tries to trick you by adding a number, but it’s just like we wrote `s = s + 3`. We know that a string “plus” anything else means to use concatenation.

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!

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



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

FIGURE 4.1 Indexing for a string

You also need to know that a `String` is immutable, or unchangeable. This means calling a method on a `String` will return a different `String` object rather than changing the value of the reference. In this chapter, you use immutable objects. In Chapter 6, "Class Design," you learn how to create immutable objects of your own.

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.

Determining the Length

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

```
public int length()
```

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

```
var name = "animals";
System.out.println(name.length());  // 7
```

Wait. It outputs `7`? Didn't we just tell you that Java counts from zero? 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.

Getting a Single Character

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

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

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

```
var name = "animals";
System.out.println(name.charAt(0));  // a
System.out.println(name.charAt(6));  // s
System.out.println(name.charAt(7));  // exception
```

Since indexes start counting with zero, `charAt(0)` returns the “first” character in the sequence. Similarly, `charAt(6)` returns the “seventh” character in the sequence. However, `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 learn more about exceptions in Chapter 11, “Exceptions and Localization.”

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

Working with Code Points

In this book and on the exam, we often use the ASCII data encoding format. Around the world some characters use a longer encoding called Unicode which has a wider range and doesn’t fit in a `char`, such as a stylized quote (’). A code point is bigger than a character, so it is expressed as a number. The relevant method signatures are as follows:

```
public int codePointAt(int index)
public int codePointBefore(int index)
public int codePointCount(int beginIndex, int endIndex)
```

The `codePointAt()` returns the numeric value of the code point at the specified index. The `codePointBefore()` method does the same, but looks at the value before the index. Finally, the `codePointCount()` method returns the number of code points between two indexes.

```
var s = "Weâ€™re done feeding the animals";
System.out.println(s.charAt(0) + " " + s.codePointAt(0));  // W
87
System.out.println(s.charAt(2) + " " + s.codePointAt(2));  //
â€™ 8217
System.out.println(s.codePointBefore(3));                 //
8217
System.out.println(s.codePointCount(0,4));                // 4
```

Don’t worry! You do not need to memorize the ASCII or Unicode values. You just need to know that if you see `codePointAt()` on the exam that it functions similarly to `charAt()` for ASCII characters, returning the numeric value of the character at the location.

Getting a Substring

The method `substring()` is similar to `charAt()` except it returns a group of characters from 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 one 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:

```
public String substring(int beginIndex)
public 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 4.2 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`.

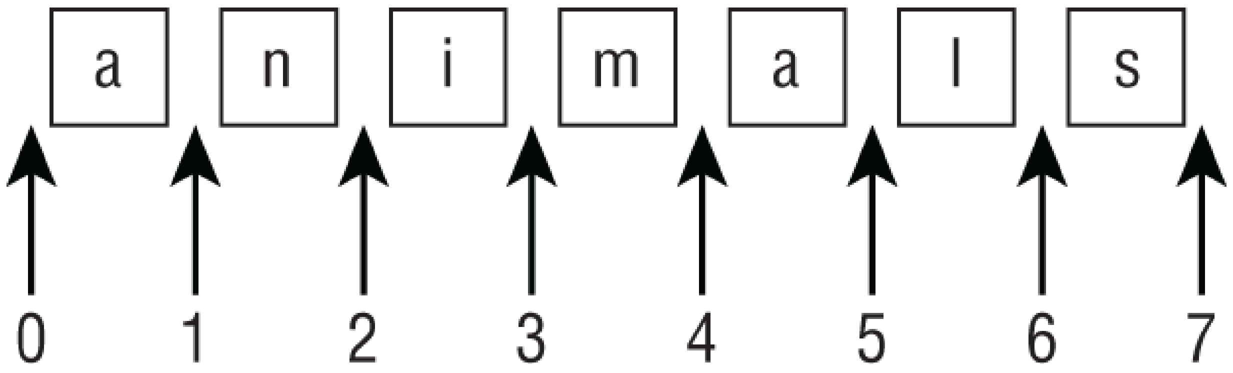


FIGURE 4.2 Indexes for a substring

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

```
var name = "animals";
System.out.println(name.substring(3));                 //
mals
System.out.println(name.substring(name.indexOf('m')));   //
mals
System.out.println(name.substring(3, 4));              // m
System.out.println(name.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`. This 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(name.substring(3, 3));  // empty string
System.out.println(name.substring(3, 2));  // exception
System.out.println(name.substring(3, 8));  // exception
```

The first example in this set prints an empty string. The request is for the characters starting with index `3` until we 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.

Finding an Index

The method `indexOf()` looks at the characters in the string and finds the first index that matches the desired value. The `indexOf` method can work with an individual character or a whole `String` as input. It can also start and end the search from specific positions. Note, the starting index is inclusive, and the ending index is exclusive. 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:

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

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

```
var name = "animals";
System.out.println(name.indexOf('a'));          // 0
System.out.println(name.indexOf("al"));         // 4
System.out.println(name.indexOf('a', 4));       // 4
System.out.println(name.indexOf("al", 5));      // -1
System.out.println(name.indexOf('a', 2, 4));    // -1
System.out.println(name.indexOf("al", 2, 6));   // 4
```

Since indexes begin with `0`, the first `'a'` matches at that position. Therefore, line 11 outputs 0. On line 12, Java looks for a more specific string, so it matches later. On line 13, Java shouldn't even look at the characters until it gets to index `4`. Line 14 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, instead returning `–1`. 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.

Line 15 looks for a match starting at index `2` and earlier than index `4`. This means indices `2` or `3`. Since neither of those matches, the method returns `-1`. Finally, line 16 looks for a match starting at index `2` since starting indexes are inclusive. It ends before at index `6` since the end index is exclusive. This means indices `2`, `3`, `4`, and `5`. The characters at index `4` and `5` match the target. The first one of those is `4`, which is returned.

Adjusting Case

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

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

The following code shows how to use these methods:

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

These methods do what they say. The `toUpperCase()` method converts any lowercase characters to uppercase in the returned string. The `toLowerCase()` method 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.

Checking for Equality

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 ignores the characters' case. The method signatures are as follows:

```
public boolean equals(Object obj)
public 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 applies only to `String` objects, so it can take the more specific type as the parameter.

In Java, `String` values are case-sensitive. That means `"abc"` and `"ABC"` are considered different values. With that in mind, 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".equals(6));                // false
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. The third example shows what happens if you pass a different type. In the last example, the values differ only by case, but it is OK because we called the method that ignores differences in case.

Real World Scenario

Overriding *toString()*, *equals(Object)*, and *hashCode()*

Knowing how to properly override `toString()`, `equals(Object)`, and `hashCode()` was part of Java certification exams in the past. As a professional Java developer, it is still important for you to know at least the basic rules for overriding each of these methods.

- `toString()`: The `toString()` method is called when you try to print an object or concatenate the object with a `String`. It is commonly overridden with a version that prints a unique description of the instance using its instance fields.
- `equals(Object)`: The `equals(Object)` method is used to compare objects, with the default implementation just using the == operator. You should override the `equals(Object)` method any time you want to conveniently compare elements for equality, especially if this requires checking numerous fields.
- `hashCode()`: Any time you override `equals(Object)`, you must override `hashCode()` to be consistent. This means that for any two objects, if `a.equals(b)` is `true`, then `a.hashCode()==b.hashCode()` must also be `true`. If they are not consistent, this could lead to invalid data and side effects in hash-based collections such as `HashMap` and `HashSet`.

All of these methods provide a default implementation in `Object`, but if you want to make intelligent use of them, you should override them.

Searching for Substrings

Often, you need to search a larger string to determine if a substring is contained within it. The `startsWith()` and `endsWith()` methods look at whether the provided value matches part of the `String`. There is also an overloaded `startsWith()` that specifies where in the `String` to start looking. The `contains()` method isn't as particular; it looks for matches anywhere in the `String`. The method signatures are as follows:

```
public boolean startsWith(String prefix)
public boolean startsWith(String prefix, int fromIndex)
public boolean endsWith(String suffix)
public boolean contains(CharSequence charSeq)
```

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".startsWith("b", 1)); // true
System.out.println("abc".startsWith("b", 2)); // false

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

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

Again, nothing surprising here. Java is doing a case-sensitive check on the values provided. Note that the `contains()` method is a convenience method so you don't have to write `str.indexOf(otherString) != -1`.

Replacing Values

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:

```
public String replace(char oldChar, char newChar)
public 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.

Removing Whitespace

These methods remove 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 does everything that `trim()` does, but it supports Unicode.

NOTE

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

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

Additionally, 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:

```
public String strip()
public String stripLeading()
public String stripTrailing()
public 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 `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`.

Working with Indentation

Now that Java supports text blocks, it is helpful to have methods that deal with indentation. Both of these are a little tricky, so read carefully!

```
public String indent(int numberSpaces)
public String stripIndent()
```

The `indent()` method adds the same number of blank spaces to the beginning of each line if you pass a positive number. If you pass a negative number, it tries to remove that number of whitespace characters from the beginning of the line. If you pass zero, the indentation will not change.

> **NOTE**
>
> If you call `indent()` with a negative number and try to remove more whitespace characters than are present at the beginning of the line, Java will remove all that it can find.

This seems straightforward enough. However, `indent()` also normalizes whitespace characters. What does *normalizing* whitespace mean, you ask? First, a line break is added to the end of the string if not already there.

Second, any line breaks are converted to the `\n` format. Regardless of whether your operating system uses `\r\n` (Windows) or `\n` (Mac/Unix), Java will standardize on `\n` for you.

The `stripIndent()` method is useful when a `String` was built with concatenation rather than using a text block. It gets rid of all incidental whitespace. This means that all nonblank lines are shifted left so the same number of whitespace characters are removed from each line and the first character that remains is not blank. Like `indent()`, `\r\n` is turned into `\n`. However, the `stripIndent()` method does not add a trailing line break if it is missing.

Well, that was a lot of rules. Table 4.1 provides a reference to make them easier to remember.

TABLE 4.1 Rules for `indent()` and `stripIndent()`

Method	**Indent change**	**Normalizes existing line breaks**	**Adds line break at end if missing**
`indent(n)` where `n > 0`	Adds `n` spaces to beginning of each line	Yes	Yes
`indent(n)` where `n == 0`	No change	Yes	Yes
`indent(n)` where `n < 0`	Removes up to `n` spaces from each line where the same number of characters is removed from each nonblank line	Yes	Yes
`stripIndent()`	Removes all leading incidental whitespace	Yes	No

The following code shows how to use these methods. Don’t worry if the results aren’t what you expect. We explain each one.

```
10: var block = """
11:             a
12:              b
```

```
13:             c""";
14: var concat = " a\n"
15:            + "  b\n"
16:            + " c";
17: System.out.println(block.length());                 // 6
18: System.out.println(concat.length());                // 9
19: System.out.println(block.indent(1).length());       // 10
20: System.out.println(concat.indent(-1).length());     // 7
21: System.out.println(concat.indent(-4).length());     // 6
22: System.out.println(concat.stripIndent().length());  // 6
```

Lines 10–16 create similar strings using a text block and a regular `String`, respectively. We say “similar” because `concat` has a whitespace character at the beginning of each line while `block` does not.

Line 17 counts the six characters in `block`, which are the three letters, the blank space before `b`, and the `\n` after `a` and `b`. Line 18 counts the nine characters in `concat`, which are the three letters, one blank space before `a`, two blank spaces before `b`, one blank space before `c`, and the `\n` after `a` and `b`. Count them up yourself. If you don’t understand which characters are counted, it will only get more confusing.

On line 19, we ask Java to add a single blank space to each of the three lines in `block`. However, the output says we added 4 characters rather than 3 since the length went from 6 to 10. This mysterious additional character is thanks to the line termination normalization. Since the text block doesn’t have a line break at the end, `indent()` adds one!

On line 20, we remove one whitespace character from each of the three lines of `concat`. This gives a length of seven. We started with nine, got rid of three characters, and added a trailing normalized new line.

On line 21, we ask Java to remove four whitespace characters from the same three lines. Since there are not four whitespace characters, Java does its best. The single space is removed before `a` and `c`. Both spaces are removed before `b`. The length of six should make sense here; we removed one more character here than on line 20.

Finally, line 22 uses the `stripIndent()` method. All of the lines have at least one whitespace character. Since they do not all have two whitespace characters, the method gets rid of only one character per line. Since no new line is added by `stripIndent()`, the length is six, which is three less than the original nine.

Checking for Empty or Blank *String*s

Java provides convenience methods for whether a `String` has a length of zero or contains only whitespace characters. The method signatures are as follows:

```
public boolean isEmpty()
public boolean isBlank()
```

The following code shows how to use these methods:

```
System.out.println(" ".isEmpty());  // false
System.out.println("".isEmpty());   // true
System.out.println(" ".isBlank());  // true
System.out.println("".isBlank());   // true
```

The first line prints `false` because the `String` is not empty; it has a blank space in it. The second line prints `true` because this time, there are no characters in the `String`. The final two lines print `true` because there are no characters other than whitespace present.

Formatting Values

There are methods to format `String` values using formatting flags. Two of the methods take the format string as a parameter, and the other uses an instance for that value. One method takes a `Locale`, which you learn about in Chapter 11.

The method parameters are used to construct a formatted `String` in a single method call, rather than via a lot of format and concatenation operations. They return a reference to the instance they are called on so that operations can be chained together. The method signatures are as follows:

```
public static String format(String format, Object… args)
public static String format(Locale loc, String format, Object…
args)
public String formatted(Object… args)
```

The following code shows how to use these methods:

```
var name = "Kate";
var orderId = 5;

// All print: Hello Kate, order 5 is ready
System.out.println("Hello "+name+", order "+orderId+" is
ready");
System.out.println(String.format("Hello %s, order %d is ready",
   name, orderId));
System.out.println("Hello %s, order %d is ready"
   .formatted(name, orderId));
```

In the `format()` and `formatted()` operations, the parameters are inserted and formatted via symbols in the order that they are provided in the vararg. Table 4.2 lists the ones you should know for the exam.

TABLE 4.2 Common formatting symbols

Symbol	**Description**
`%s`	Applies to any type, commonly `String` values
`%d`	Applies to integer values like `int` and `long`
`%f`	Applies to floating-point values like `float` and `double`
`%n`	Inserts a line break using the system-dependent line separator

The following example uses all four symbols from Table 4.2:

```
var name = "James";
var score = 90.25;
var total = 100;
System.out.println("%s:%n   Score: %f out of %d"
   .formatted(name, score, total));
```

This prints the following:

```
James:
   Score: 90.250000 out of 100
```

Mixing data types may cause exceptions at runtime. For example, the following throws an exception because a floating-point number is used when an integer value is expected:

```
var str = "Food: %d tons".formatted(2.0); //
IllegalFormatConversionException
```

Using *format()* with Flags

Besides supporting symbols, Java also supports optional flags between the `%` and the symbol character. In the previous example, the floating-point number was printed as `90.250000`. By default, `%f` displays exactly six digits past the decimal. If you want to display only one digit after the decimal, you can use `%.1f` instead of `%f`. The `format()` method relies on rounding rather than truncating when shortening numbers. For example, `90.250000` will be displayed as `90.3` (not `90.2`) when passed to `format()` with `%.1f`.

The `format()` method also supports two additional features. You can specify the total length of output by using a number before the decimal symbol. By default, the method will fill the empty space with blank spaces. You can also fill the empty space with zeros by placing a single zero before the decimal symbol. The following examples use brackets, `[]`, to show the start/end of the formatted value:

```
var pi = 3.14159265359;
System.out.format("[%f]",pi);      // [3.141593]
System.out.format("[%12.8f]",pi);  // [  3.14159265]
System.out.format("[%012f]",pi);   // [00003.141593]
System.out.format("[%12.2f]",pi);  // [        3.14]
System.out.format("[%.3f]",pi);    // [3.142]
```

The `format()` method supports a lot of other symbols and flags. You don't need to know any of them for the exam beyond what we've discussed already.

Method Chaining

Ready to put together everything you just learned about? It is common to call multiple methods, as shown here:

```
var start = "AniMaL   ";
var trimmed = start.trim();                 // "AniMaL"
var lowercase = trimmed.toLowerCase();      // "animal"
var 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.

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 none of the code on lines 6 and 7 changes `a`, the value remains `"abc"`.

However, `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 objects 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 learn about in Chapter 13, “Concurrency.” `StringBuffer` is not on the exam. It performs slower than `StringBuilder`, so just use `StringBuilder`.

In this section, we 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 (a blank plus adding each letter in the alphabet). The exam will likely try to trick you with respect to `StringBuilder` being mutable and `String` being immutable.

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`. This 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()` is 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. The first two examples tell 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.

Using Common Methods

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

```
var 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. The `substring()` method returns the `String` starting with index `0` and ending right before index `4`.

The `length()` method 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 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. The `substring()` method is really just a method that inquires about the state of the `StringBuilder`.

Appending Values

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:

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

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

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

Nice method chaining, isn't it? The `append()` method 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.

Applying Code Points

The `codePointAt()`, `codePointBefore()`, and `codePointCount()` methods from `String` are also available on `StringBuilder`. There's one more method you need to know for code points that is only on `StringBuilder`:

```
public StringBuilder appendCodePoint(int codePoint)
```

It works like the `append()` method in the previous section except it takes an integer representing the Unicode value, converts it to a character, and appends it to the `StringBuilder`.

```
var sb = new StringBuilder()
   .appendCodePoint(87).append(',')
   .append((char)87).append(',')
   .append(87).append(',')
   .appendCodePoint(8217);
System.out.println(sb);  // W,W,87,â€™
```

Like we saw with `String`, it also handles non-ASCII characters like a stylized quote (`′`). Again, you do not need to know the numeric values for characters for the exam, but you should understand how the text in this example is generated.

Inserting Data

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:

```
public 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: var 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 using available writing materials so you won't be confused.

Deleting Contents

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:

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

The following code shows how to use these methods:

```
var sb = new StringBuilder("abcdef");
sb.delete(1, 3);                  // sb = adef
sb.deleteCharAt(5);               // 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:

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

Replacing Portions

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

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

The following code shows how to use this method:

```
var 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 the value `"sty"` in that position.

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

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

It 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.

Reversing

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:

```
public StringBuilder reverse()
```

The following code shows how to use this method:

```
var 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.

Working with *toString()*

The `Object` class contains a `toString()` method that many classes provide custom implementations of. The `StringBuilder` class is one of these.

The following code shows how to use this method:

```
var 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 2, you learned how to use == to compare numbers and that object references refer to the same object. Earlier in this chapter, we saw the `equals()` method on `String`. In this section, we look at what it means for two objects to be equivalent or the same. We also look at the impact of the `String` pool on equality.

Comparing *equals()* and ==

Consider the following code that uses == with objects:

```
var one = new StringBuilder();
var two = new StringBuilder();
var 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. The `one` and `two` variables are both completely separate `StringBuilder` objects, giving us two objects. Therefore, the first print statement gives us `false`. The `three` variable 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 `equals()` uses logical equality rather than object equality for `String` objects.

```
var x = "Hello World";
var 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.

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

```
var name = "a";
var builder = new StringBuilder("a");

System.out.println(name == 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. The `myObject.toString()` method returns 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.

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

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

```
var x = "Hello World";
var 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?

```
var singleString = "hello world";
var concat = "hello ";
concat += "world";
System.out.println(singleString == concat); // false
```

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

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

The first 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.

`public String intern()`

If the literal is not yet in the string pool, Java will add it at this time.

```
var name = "Hello World";
var 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.

```
var first = "rat" + 1;
var second = "r" + "a" + "t" + "1";
var 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 lines 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`.

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

Understanding 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. The `char` type 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 look at creating an array of primitives and objects, sorting, searching, and varargs.

Creating an Array of Primitives

Figure 4.3 shows the most common way to create an array. It specifies the type of the array (`int`) and the size (`3`). The brackets tell you this is an array.

A sample structure of an array. It reads int as type of array, square braces as array symbol, and three in braces as size of array.

FIGURE 4.3 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 1, the default value of an `int` is `0`. Since `numbers` is a reference variable, it points to the array object, as shown in Figure 4.4. 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`.

A sample structure of an empty array. An arrow mark points out the first cell indicating the entry of numbers. Thr first row has the entry of index entries 0, 1, and 2 followed by entry of elements entries 0 in the second row. The index row is highlighted.

FIGURE 4.4 An empty array

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

```
int[] moreNumbers = 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 4.5 shows what this array looks like.

A sample structure of an initialized array. An arrow mark points out the first cell indicating the entry of morenumbers. The first row has the entry of index entries 0, 1, and 2 followed by entry of elements entries 42, 55, and 99 in the second row. The index row is highlighted.

FIGURE 4.5 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 equals 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[] moreNumbers = {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`:

```
String[] bugs = { "cricket", "beetle", "ladybug" };
String[] alias = bugs;
String[] anotherArray = { "cricket", "beetle", "ladybug" };
System.out.println(bugs.equals(alias));        // true
System.out.println(bugs.equals(anotherArray)); // false
System.out.println(bugs.toString());           //
[Ljava.lang.String;@160bc7c0
```

We can call `equals()` because an array is an object. The first test with `alias` returns `true` because of reference equality. Why does the second equality test return `false`? The `equals()` method on arrays does not look at the elements of the array.

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.

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

We can see our `bugs` array represented in memory in Figure 4.6. Make sure you understand this figure. The array does not allocate space for the `String` objects. Instead, it allocates space for a reference to where the objects are really stored.

A sample structure of an array pointing to strings. An arrow mark points out the first cell indicating the enrty of bugs. The entries depict 0 as cricket, 1 as beetle, and 2 as ladybug.

FIGURE 4.6 An array pointing to strings

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

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

You got us. It was a review of Chapter 1 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?

```
public 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[]`allows only `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 this line will compile and 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 that elements 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. Watch out for compiler errors like the following on the exam!

```
4: String[] mammals = {"monkey", "chimp", "donkey"};
5: System.out.println(mammals.length());           // DOES NOT
COMPILE
```

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

```
4: var birds = new String[6];
5: 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. The `length` attribute does not consider what is in the array; it considers only 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: var numbers = new int[10];
6: for (int i = 0; i < numbers.length; i++)
7:    numbers[i] = i + 5;
8: for(int n : numbers)
9:    System.out.println(n);
```

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` to the index of the element plus 5. Lines 8 and 9 print the numbers in the array, using the for-each loop that you learned about in Chapter 3, “Making Decisions.”

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?

```
3: var numbers = new int[10];
4: numbers[10] = 3;
5:
6: numbers[numbers.length] = 5;
7:
```

```
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 that `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 tenth index.

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` 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, you can assume the necessary imports are there. 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 s : strings)
   System.out.print(s + " ");
```

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 Chapter 9, "Collections and Generics," you learn how to create custom sort orders using something called a *comparator*.

You can use 7Up, the soda, to help remember the order. Numbers (7) sort first, followed by uppercase (U), and then lowercase (p).

Searching

Java also provides a convenient way to search, but only if the array is already sorted. Table 4.3 covers the rules for binary search.

TABLE 4.3 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 is undefined

Let's try these rules with an example:

```
int[] numbers = {2,4,6,8};
System.out.println(Arrays.binarySearch(numbers, 2)); // 0
System.out.println(Arrays.binarySearch(numbers, 4)); // 1
System.out.println(Arrays.binarySearch(numbers, 1)); // -1
System.out.println(Arrays.binarySearch(numbers, 3)); // -2
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 defined. 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 cover the `equals() and compare()` methods, and then we go on to `mismatch()`. These methods are overloaded to take a variety of parameters.

Using *equals()*

While == compares object references, `Arrays` includes overloaded versions of `equals()` that lets you check if the arrays are the same size and contain the same elements, in the same order. For example:

```
System.out.println(new int[] {1} == new int[] {1});
// false
System.out.println(Arrays.equals(new int[] {1}, new int[]
{1}));    // true
System.out.println(Arrays.equals(new int[] {1}, new int[]
{2}));    // false
System.out.println(Arrays.equals(new int[] {1}, new int[] {1,
2})); // false
```

When comparing elements, it uses == for primitive values and `equals()` for object values.

Using *compare()*

There are a bunch of rules you need to know before calling `compare()`. Luckily, these are the same rules you need to know in Chapter 9 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 see in Chapter 8, “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 4.4 shows examples of these rules in action.

TABLE 4.4 `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
```

Using *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 4.5. If you don't understand why all of the values are there, please go back and study this section again.

TABLE 4.5 Equality vs. comparison vs. mismatch

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

Using Methods with 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. You learn how to call a method using varargs in Chapter 5, "Methods." 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.

Working with Arrays of 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 an Array of Arrays

Multiple array separators are all it takes to declare arrays of arrays. While they aren't really multidimensional, it helps to think of them as such. 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 [][]; // 2D and 3D arrays
```

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 array and the array it contains 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 4.7. 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`. You start at index `0` in the top array. Then you go to index `1` in the next array.

A sample structure of a sparesly populated array of arrays. An arrow mark points out the first cell indicating the rectangle. The entries are 0, 1, and 2. The arrows from each entries points out two elements set 0 and 1 in a rectangular shape.

FIGURE 4.7 A sparsely populated array of arrays

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 4.8. This time the array is of primitives, so they are shown as if they are in the array themselves.

A sample structure of an asymmetric array of arrays. An arrow mark points out the first cell indicating the different sizes. The entries are 0, 1, and 2. The arrows from each entries point out asymmetric array, four elements from entry 0, two elements from entry 1, and six elements from enrty 3.

FIGURE 4.8 An asymmetric array of arrays

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[2][];
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, could look like a matrix.

Using an Array of Arrays

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

```
var 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 top-level array 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.

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 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. Additionally, we cover the `BigInteger` and `BigDecimal` classes in this section.

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

Finding the Minimum and Maximum

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

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

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static 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.

Rounding Numbers

The `Math.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 `Math.round()` are as follows:

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

There are two overloaded methods to ensure that 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`.

Determining the Ceiling and Floor

The `Math.ceil()` method takes a `double` value. If it is a whole number, it returns the same value. If it has any fractional value, it rounds up to the next whole number. By contrast, the `Math.floor()` method discards any values after the decimal.

The method signatures are as follows:

```
public static double ceil(double num)
public static double floor(double num)
```

The following shows how to use these methods:

```
double c = Math.ceil(3.14);  // 4.0
double f = Math.floor(3.14); // 3.0
```

The first line returns `4.0` because four is the integer, just larger. The second line returns `3.0` because it is the integer, just smaller.

Calculating Exponents

The `Math.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 0.5 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:

```
public static 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.

Generating Random Numbers

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

```
public static 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.

While not on the exam, it is common to use the `Random` class for generating pseudo-random numbers. It allows generating numbers of different types.

Using BigInteger and BigDecimal

All of the `Math` APIs in the previous section used java primitive types. However, these are not always precise enough. Especially when dealing with money or large numbers. Luckily, Java has built in classes called `BigInteger` and `BigDecimal` that can handle values that don't fit into the primitive numeric types. Like `String`, these classes are immutable so you chain methods to perform multiple operations.

While there are constructors, it is recommended to use the `valueOf()` method where possible. Note that you can pass a `long` to either type, but a `double` only to `BigDecimal`.

```
var bigInt = BigInteger.valueOf(5_000L);
var bigDecimal = BigDecimal.valueOf(5_000L);
bigDecimal = BigDecimal.valueOf(5_000.00);
```

Both classes provide constants for the most common values like `BigInteger.ZERO` and `BigDecimal.ONE`.

There are methods to perform math operations with these types, such as:

```
var bigInt = BigInteger.valueOf(199)
   .add(BigInteger.valueOf(1))
   .divide(BigInteger.TEN)
   .max(BigInteger.valueOf(6));
System.out.println(bigInt);  // 20
```

This example starts by adding 199 and 1 which gives 200. It then divides by 10 resulting in 20. Finally, `max()` sees that 20 is larger than 6 and we have the result.

Real World Scenario

When to use BigInteger and BigDecimal

In the real world, you would use `BigInteger` to handle integer values that don't fit within `int` or `long`, such as in the following example.

```
System.out.println(new BigInteger("12345123451234512345"));
System.out.println(12345123451234512345L);  // DOES NOT
COMPILE
```

Likewise, `BigDecimal` is for larger values that don't fit within `float` or `double`. It is often used for small values that involve money. Why? Well, sometimes floating point numbers are stored in memory in unexpected ways. Consider the following example.

```
double amountInCents1 = 64.1 * 100;
System.out.println(amountInCents1);  // 6409.999999999999
```

The difference between the expected value and actual value is referred to as the floating-point error. Oftentimes, these errors don't significantly change the result of an operation, but it would be bad to do so when working with money! We can fix this by using `BigDecimal` instead.

```
BigDecimal amountInCents2 = BigDecimal.valueOf(64.1)
   .multiply(BigDecimal.valueOf(100));
System.out.println(amountInCents2);  // 6410.0
```

Working with Dates and Times

Java provides a number of APIs for working with dates and times. There's also an old `java.util.Date` class, but it is not on the exam. You need an import statement to work with the modern date and time classes. To use it, add this `import` to your program:

```
import java.time.*;    // import time classes
```

Day vs. Date

In American English, the word *date* is used to represent two different concepts. Sometimes, it is the month/day/year combination when something happened, such as January 1, 2025. Sometimes, it is the day of the month, such as "Today's date is the 6th."

That's right—the words *day* and *date* are often used as synonyms. Be alert to this on the exam, especially if you live someplace where people are more precise about this distinction.

In the following sections, we look at creating and manipulating dates and times, including time zones and daylight saving time.

Creating Dates and Times

In the real world, we usually talk about dates and time zones as if the other person is located near us. For example, if you say to me, "I'll call you at 11 on Tuesday morning," we assume that 11 means the same thing to both of us. But if I live in New York and you live in California, we need to be more specific. California is three hours earlier than New York because the states are in different time zones. You would instead say, "I'll call you at 11 EST (Eastern Standard Time) on Tuesday morning."

When working with dates and times, the first thing to do is to decide how much information you need. The exam gives you four choices.

`LocalDate` Contains just a date—no time and no time zone. A good example of `LocalDate` is your birthday this year. It is your birthday for a full day, regardless of what time it is.

`LocalTime` Contains just a time—no date and no time zone. A good example of `LocalTime` is midnight. It is midnight at the same time every day.

`LocalDateTime` Contains both a date and time but no time zone. A good example of `LocalDateTime` is “the stroke of midnight on New Year’s Eve.”

`ZonedDateTime` Contains a date, time, and time zone. A good example of `ZonedDateTime` is “a conference call at 9 a.m. EST.” If you live in California, you’ll have to get up really early since the call is at 6 a.m. local time!

You obtain date and time instances using a `static` method.

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
System.out.println(ZonedDateTime.now());
```

Each of the four classes has a `static` method called `now()`, which gives the current date and time. Your output is going to depend on the date/time when you run it and where you live. The authors live in the United States, making the output look like the following when run on July 25 at 9:13 a.m.:

```
2025-07-25
09:13:07.768
2025-07-25T09:13:07.768
2025-07-25T09:13:07.769-04:00[America/New_York]
```

The key is the type of information in the output. The first line contains only a date and no time. The second contains only a time and no date. The time displays hours, minutes, seconds, and fractional seconds. The third contains both a date and a time. The output uses `T` to separate the date and time when converting `LocalDateTime` to a `String`. Finally, the fourth adds the time zone offset and time zone. New York is four time zones away from Greenwich Mean Time (GMT).

Greenwich Mean Time is a time zone in Europe that is used as time zone zero when discussing offsets. You might have also heard of *Coordinated Universal Time*, which is a time zone standard. It is abbreviated as UTC, as a compromise between the English and French names. (That’s not a typo.

UTC isn’t actually the proper acronym in either language!) UTC uses the same time zone zero as GMT.

First, let’s try to figure out how far apart the following moments are in time. Notice how India has a half-hour offset, not a full hour. To approach a problem like this, you subtract the time zone from the time. This gives you the GMT equivalent of the time:

```
2025–06–20T06:50+05:30[Asia/Kolkata]    // GMT 2025–06–20 01:20
2025–06–20T07:50-04:00[US/Eastern]      // GMT 2025–06–20 11:50
```

Remember that you need to add when subtracting a negative number. After converting to GMT, you can see that the U.S. Eastern time occurs 10 and a half hours after the Kolkata time.

NOTE

The time zone offset can be listed in different ways: +02:00, GMT+2, and UTC+2 all mean the same thing. You might see any of them on the exam.

If you have trouble remembering this, try to memorize one example where the time zones are a few zones apart, and remember the direction. In the United States, most people know that the East Coast is three hours ahead of the West Coast. And most people know that Asia is ahead of Europe. Just don’t cross time zone zero in the example that you choose to remember. The calculation works the same way, but it isn’t as great a memory aid.

Wait, I Don't Live in the United States

The exam recognizes that exam takers live all over the world, and it will not ask you about the details of U.S. date and time formats. That said, our examples do use U.S. date and time formats, as will the questions on the exam. Just remember that the month comes before the date. Also, Java tends to use a 24-hour clock even though the United States uses a 12-hour clock with a.m./p.m.

Now that you know how to create the current date and time, let's look at other specific dates and times. To begin, let's create just a date with no time. Both of these examples create the same date:

```
var date1 = LocalDate.of(2025, Month.JANUARY, 20);
var date2 = LocalDate.of(2025, 1, 20);
```

Both pass in the year, month, and date. Although it is good to use the `Month` constants (to make the code easier to read), you can pass the `int` number of the month directly. Just use the number of the month the same way you would if you were writing the date in real life.

The method signatures are as follows:

```
public static LocalDate of(int year, int month, int dayOfMonth)
public static LocalDate of(int year, Month month, int
dayOfMonth)
```

NOTE

Up to now, we've been continually telling you that Java counts starting with 0. Well, months are an exception. For months in the new date and time methods, Java counts starting from 1, just as we humans do.

When creating a time, you can choose how detailed you want to be. You can specify just the hour and minute, or you can include the number of seconds. You can even include nanoseconds if you want to be very precise. (A nanosecond is a billionth of a second, although you probably won't need to be that specific.)

```
var time1 = LocalTime.of(6, 15);               // hour and
minute
var time2 = LocalTime.of(6, 15, 30);           // + seconds
var time3 = LocalTime.of(6, 15, 30, 200);      // + nanoseconds
```

These three times are all different but within a minute of each other. The method signatures are as follows:

```
public static LocalTime of(int hour, int minute)
public static LocalTime of(int hour, int minute, int second)
public static LocalTime of(int hour, int minute, int second,
int nanos)
```

You can combine dates and times into one object.

```
var dateTime1 = LocalDateTime.of(2025, Month.JANUARY, 20, 6,
15, 30);
var dateTime2 = LocalDateTime.of(date1, time2);
```

The first line of code shows how you can specify all of the information about the `LocalDateTime` right in the same line. The second line of code shows how you can create `LocalDate` and `LocalTime` objects separately first and then combine them to create a `LocalDateTime` object.

There are a lot of method signatures since there are more combinations. The following method signatures use integer values:

```
public static LocalDateTime of(int year, int month,
       int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, int month,
       int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, int month,
       int dayOfMonth, int hour, int minute, int second, int
nanos)
```

Others take a `Month` reference:

```
public static LocalDateTime of(int year, Month month,
       int dayOfMonth, int hour, int minute)
```

```
public static LocalDateTime of(int year, Month month,
       int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, Month month,
       int dayOfMonth, int hour, int minute, int second, int
nanos)
```

Finally, one takes an existing `LocalDate` and `LocalTime`:

```
public static LocalDateTime of(LocalDate date, LocalTime time)
```

To create a `ZonedDateTime`, we first need to get the desired time zone. We will use `US/Eastern` in our examples:

```
var zone = ZoneId.of("US/Eastern");
var zoned1 = ZonedDateTime.of(2025, 1, 20,
      6, 15, 30, 200, zone);
var zoned2 = ZonedDateTime.of(date1, time1, zone);
var zoned3 = ZonedDateTime.of(dateTime1, zone);
```

We start by getting the time zone object. Then we use one of three approaches to create the `ZonedDateTime`. The first passes all of the fields individually. We don’t recommend this approach—there are too many numbers, and it is hard to read. A better approach is to pass a `LocalDate` object and a `LocalTime` object, or a `LocalDateTime` object.

Although there are other ways of creating a `ZonedDateTime`, you only need to know three for the exam:

```
public static ZonedDateTime of(int year, int month,
   int dayOfMonth, int hour, int minute, int second,
   int nanos, ZoneId zone)
public static ZonedDateTime of(LocalDate date, LocalTime time,
   ZoneId zone)
public static ZonedDateTime of(LocalDateTime dateTime, ZoneId
zone)
```

Notice that there isn’t an option to pass in the `Month` enum. Also, we did not use a constructor in any of the examples. The date and time classes have private constructors along with `static` methods that return instances. This is known as the *factory pattern*. The exam creators may throw something like this at you:

```
var d = new LocalDate(); // DOES NOT COMPILE
```

Don’t fall for this. You are not allowed to construct a date or time object directly.

Another trick is what happens when you pass invalid numbers to `of()`, for example:

```
var d = LocalDate.of(2025, Month.JANUARY, 32); //
DateTimeException
```

You don’t need to know the exact exception that’s thrown, but it’s a clear one:

```
java.time.DateTimeException: Invalid value for DayOfMonth
       (valid values 1-28/31): 32
```

Manipulating Dates and Times

Adding to a date is easy. The date and time classes are immutable. Remember to assign the results of these methods to a reference variable so they are not lost.

```
12: var date = LocalDate.of(2025, Month.JANUARY, 20);
13: System.out.println(date);    // 2025–01–20
14: date = date.plusDays(2);
15: System.out.println(date);    // 2025–01–22
16: date = date.plusWeeks(1);
17: System.out.println(date);    // 2025–01–29
18: date = date.plusMonths(1);
19: System.out.println(date);    // 2025–02–28
20: date = date.plusYears(5);
21: System.out.println(date);    // 2030–02–28
```

This code is nice because it does just what it looks like. We start out with January 20, 2025. On line 14, we add two days to it and reassign it to our reference variable. On line 16, we add a week. This method allows us to write clearer code than `plusDays(7)`. Now `date` is January 29, 2025. On line 18, we add a month. This would bring us to February 29, 2025. However, 2025 is not a leap year (2020 and 2024 are leap years). Java is smart enough to realize that February 29, 2025, does not exist, and it gives us February 28, 2025, instead. Finally, line 20 adds five years.

NOTE

February 29 exists only in a leap year. Leap years are years that are a multiple of 4 or 400, but not other multiples of 100. For example, 2000 and 2028 are leap years, but 2100 is not.

There are also nice, easy methods to go backward in time. This time, let's work with `LocalDateTime`:

```
22: var date = LocalDate.of(2025, Month.JANUARY, 20);
23: var time = LocalTime.of(5, 15);
24: var dateTime = LocalDateTime.of(date, time);
25: System.out.println(dateTime);       // 2025-01-20T05:15
26: dateTime = dateTime.minusDays(1);
27: System.out.println(dateTime);       // 2025-01-19T05:15
28: dateTime = dateTime.minusHours(10);
29: System.out.println(dateTime);       // 2025-01-18T19:15
30: dateTime = dateTime.minusSeconds(30);
31: System.out.println(dateTime);       // 2025-01-18T19:14:30
```

Line 25 prints the original date of January 20, 2025, at 5:15 a.m. Line 26 subtracts a full day, bringing us to January 19, 2025, at 5:15 a.m. Line 28 subtracts 10 hours, showing that the date will change if the hours cause it to adjust, and it brings us to January 18, 2025, at 19:15 (7:15 p.m.). Finally, line 30 subtracts 30 seconds. You can see that all of a sudden, the display value starts showing seconds. Java is smart enough to hide the seconds and nanoseconds when we aren't using them.

It is common for date and time methods to be chained. For example, without the print statements, the previous example could be rewritten as follows:

```
var date = LocalDate.of(2025, Month.JANUARY, 20);
var time = LocalTime.of(5, 15);
var dateTime = LocalDateTime.of(date, time)
       .minusDays(1).minusHours(10).minusSeconds(30);
```

When you have a lot of manipulations to make, this chaining comes in handy. There are two ways that the exam creators can try to trick you. What do you think this prints?

```
var date = LocalDate.of(2025, Month.JANUARY, 20);
date.plusDays(10);
System.out.println(date);
```

It prints `2025-01-20`. Adding 10 days was useless because the program ignored the result. Whenever you see immutable types, pay attention to make sure that the return value of a method call isn't ignored. The exam also may test to see if you remember what each of the date and time objects includes. Do you see what is wrong here?

```
var date = LocalDate.of(2025, Month.JANUARY, 20);
date = date.plusMinutes(1);       // DOES NOT COMPILE
```

`LocalDate` does not contain time. This means you cannot add minutes to it. This can be tricky in a chained sequence of addition/subtraction operations, so make sure you know which methods in Table 4.6 can be called on which types.

TABLE 4.6 Methods in `LocalDate`, `LocalTime`, `LocalDateTime`, and `ZonedDateTime`

	Can call on `LocalDate`?	**Can call on `LocalTime`?**	**Can call on `LocalDateTime` or `ZonedDateTime`?**
`plusYears()` `minusYears()` `withYear()` `withDayOfYear()`	Yes	No	Yes
`plusMonths()` `minusMonths()` `withMonth()`	Yes	No	Yes
`plusWeeks()` `minusWeeks()`	Yes	No	Yes
`plusDays()` `minusDays()` `withDayOfMonth()`	Yes	No	Yes
`plusHours()` `minusHours()` `withHour()`	No	Yes	Yes
`plusMinutes()` `minusMinutes()` `withMinute()`	No	Yes	Yes
`plusSeconds()` `minusSeconds()` `withSecond()`	No	Yes	Yes
`plusNanos()` `minusNanos()` `withNano()`	No	Yes	Yes

Table 4.6 also includes methods that you can use to create a copy of an object with specific field(s) altered to the specified value. For example:

```
var date = LocalDate.of(2025, Month.FEBRUARY, 20);  // 2025-02-
20
```

```
var differentDay = date.withDayOfMonth(15);          // 2025-02-
15
var differentMonth = date.withDayOfYear(3);          // 2025-01-
03
var allChanged = date.withYear(2026)
   .withMonth(4)
   .withDayOfMonth(10);                              // 2026-04-
10
```

Finally, there are methods to convert from one type to another. For example:

```
var date = LocalDate.of(2025, Month.MARCH, 3);
var withTime = date.atTime(5, 30);  // 2025-03-03T05:30
var start = date.atStartOfDay();    // 2025-03-03T00:00
```

The `at_________()` methods combine the instance variable and the parameter into one new object. They are listed in Table 4.7.

TABLE 4.7 Conversion methods in `LocalDate`, `LocalTime`, and `LocalDateTime`

`LocalDate` **to** `LocalDateTime`	`atStartOfDay()`
	`atTime(int hour, int minute)`
	`atTime(int hour, int minute, int second)`
	`atTime(int hour, int minute, int second, int nanos)`
	`atTime(LocalTime time)`

`LocalTime` **to** `LocalDate`	`atDate(LocalDate date)`
`LocalDateTime` **to** `ZonedDateTime`	`atZone(ZoneId zoneId)`

Working with Periods

Now you know enough to do something fun with dates! Our zoo performs animal enrichment activities to give the animals something enjoyable to do. The head zookeeper has decided to switch the toys every month. This system will continue for three months to see how it works out.

```
public static void main(String[] args) {
   var start = LocalDate.of(2025, Month.JANUARY, 1);
   var end = LocalDate.of(2025, Month.MARCH, 30);
   performAnimalEnrichment(start, end);
}
private static void performAnimalEnrichment(LocalDate start,
LocalDate end) {
   var upTo = start;
   while (upTo.isBefore(end)) {  // check if still before end
      System.out.println("give new toy: " + upTo);
      upTo = upTo.plusMonths(1); // add a month
} }
```

This code works fine. It adds a month to the date until it hits the end date. The problem is that this method can't be reused. Our zookeeper wants to try different schedules to see which works best.

`LocalDate` and `LocalDateTime` have a method to convert themselves into `long` values, equivalent to the number of milliseconds that have passed since January 1, 1970, referred to as the *epoch*. What's special about this date? That's what Unix started using for date standards, so Java reused it.

Luckily, Java has a `Period` class that we can pass in. This code does the same thing as the previous example:

```
public static void main(String[] args) {
   var start = LocalDate.of(2025, Month.JANUARY, 1);
   var end = LocalDate.of(2025, Month.MARCH, 30);
   var period = Period.ofMonths(1);    // create a period
   performAnimalEnrichment(start, end, period);
}
private static void performAnimalEnrichment(LocalDate start,
LocalDate end,
   Period period) {             // uses the generic period

   var upTo = start;
   while (upTo.isBefore(end)) {
      System.out.println("give new toy: " + upTo);
      upTo = upTo.plus(period); // adds the period
} }
```

The method can add an arbitrary period of time that is passed in. This allows us to reuse the same method for different periods of time as our zookeeper changes their mind.

A `Period` can be positive (forward in time) or negative (backwards in time.) There are five ways to create a `Period` class.

```
var annually = Period.ofYears(1);             // every 1 year
var quarterly = Period.ofMonths(3);           // every 3 months
var everyThreeWeeks = Period.ofWeeks(-3);     // every 3 weeks
going backwards
var everyOtherDay = Period.ofDays(2);         // every 2 days
var everyYearAndAWeek = Period.of(1, 0, 7);  // every year plus
1 week
```

There's one catch. You cannot chain methods when creating a `Period`. The following code looks like it is equivalent to the `everyYearAndAWeek` example, but it's not. Only the last method is used because the methods are `static` methods.

```
var wrong = Period.ofYears(1).ofWeeks(1); // every week
```

This tricky code is really like writing the following:

```
var wrong = Period.ofYears(1);
wrong = Period.ofWeeks(1);
```

This is clearly not what you intended! That's why the `of()` method allows you to pass in the number of years, months, and days. They are all included in the same period. You will get a compiler warning about this. Compiler warnings tell you that something is wrong or suspicious without failing compilation.

The `of()` method takes only years, months, and days. The ability to use another factory method to pass weeks is merely a convenience. As you might imagine, the actual period is stored in terms of years, months, and days. When you print out the value, Java displays any nonzero parts using the format shown in Figure 4.9.

A sample structure of a period format. The expression reads system.out,println(period.of (1,2,3)); and the entries depicts P as period, 1 Y as year, 1 M as month, and 3 D as days.

FIGURE 4.9 Period format

As you can see, the `P` always starts out the `String` to show it is a period measure. Then come the number of years, number of months, and number of days. If any of these are zero, they are omitted.

Can you figure out what this outputs?

```
System.out.println(Period.ofMonths(3));
```

The output is `P3M`. Remember that Java omits any measures that are zero. You can also create a period by getting the amount of time between two `LocalDate` objects:

```
var xmas = LocalDate.of(2025, Month.DECEMBER, 25);
var newYears = LocalDate.of(2026, Month.JANUARY, 1);

System.out.println(Period.between(xmas, newYears));  // P7D
System.out.println(Period.between(newYears, xmas));  // P-7D
```

Notice how order matters. The first time `Period.between()` returns a period representing seven days, but the second time it returns a period of negative seven days.

The last thing to know about `Period` is what objects it can be used with. Let’s look at some code:

```
3:  var date = LocalDate.of(2025, 3, 20);
4:  var time = LocalTime.of(6, 15);
5:  var dateTime = LocalDateTime.of(date, time);
6:  var period = Period.ofMonths(-1);
7:  System.out.println(date.plus(period));     // 2025–02–20
8:  System.out.println(dateTime.plus(period)); // 2025–02–
20T06:15
9:  System.out.println(time.plus(period));     // Exception
```

Lines 7 and 8 work as expected. They subtract a month from March 20, 2025, giving us February 20, 2025. The first has only the date, and the second has both the date and time.

Line 9 attempts to add a month to an object that has only a time. This won't work. Java throws an `UnsupportedTemporalTypeException` and complains that we attempted to use an `Unsupported unit: Months`.

As you can see, you have to pay attention to the type of date and time objects every place you see them.

Working with Durations

You've probably noticed by now that a `Period` is a day or more of time. There is also `Duration`, which is intended for smaller units of time. For `Duration`, you can specify the number of days, hours, minutes, seconds, or nanoseconds. And yes, you could pass 365 days to make a year, but you really shouldn't—that's what `Period` is for.

Conveniently, `Duration` works roughly the same way as `Period`, except it is used with objects that have time. `Duration` is output beginning with `PT`, which you can think of as a period of time. A `Duration` is stored in hours, minutes, and seconds. The number of seconds includes fractional seconds.

We can create a `Duration` using a number of different granularities:

```
var daily = Duration.ofDays(1);                // PT24H
var hourly = Duration.ofHours(1);              // PT1H
var everyMinute = Duration.ofMinutes(1);       // PT1M
var everyTenSeconds = Duration.ofSeconds(10); // PT10S
var everyMilli = Duration.ofMillis(1);         // PT0.001S
var everyNano = Duration.ofNanos(1);           // PT0.000000001S
```

`Duration` doesn't have a factory method that takes multiple units like `Period` does. If you want something to happen every hour and a half, you specify 90 minutes.

`Duration` includes another more generic factory method. It takes a number and a `TemporalUnit`. The idea is, say, something like "5 seconds." However, `TemporalUnit` is an interface. At the moment, there is only one implementation named `ChronoUnit`.

The previous example could be rewritten like this:

```
var daily = Duration.of(1, ChronoUnit.DAYS);
var hourly = Duration.of(1, ChronoUnit.HOURS);
var everyMinute = Duration.of(1, ChronoUnit.MINUTES);
var everyTenSeconds = Duration.of(10, ChronoUnit.SECONDS);
var everyMilli = Duration.of(1, ChronoUnit.MILLIS);
var everyNano = Duration.of(1, ChronoUnit.NANOS);
```

`ChronoUnit` also includes some convenient units such as `ChronoUnit.HALF_DAYS` to represent 12 hours.

ChronoUnit for Differences

`ChronoUnit` is a great way to determine how far apart two `Temporal` values are. `Temporal` includes `LocalDate`, `LocalTime`, and so on. `ChronoUnit` is in the `java.time.temporal` package.

```
var one = LocalTime.of(5, 15);
var two = LocalTime.of(6, 55);
var date = LocalDate.of(2025, 1, 20);
System.out.println(ChronoUnit.HOURS.between(one, two));
// 1
System.out.println(ChronoUnit.MINUTES.between(one, two));
// 100
System.out.println(ChronoUnit.MINUTES.between(one, date));
// DateTimeException
```

The first print statement shows that `between` truncates rather than rounds. The second shows how easy it is to count in different units. Just change the `ChronoUnit` type. The last reminds us that Java will throw an exception if we mix up what can be done on date versus time objects.

Alternatively, you can truncate any object with a time element. For example:

```
LocalTime time = LocalTime.of(3,12,45);
System.out.println(time);      // 03:12:45
LocalTime truncated = time.truncatedTo(ChronoUnit.MINUTES);
System.out.println(truncated); // 03:12
```

This example zeroes out any fields smaller than minutes. In our case, it gets rid of the seconds.

Using a `Duration` works the same way as using a `Period`. For example:

```
7:  var date = LocalDate.of(2025, 1, 20);
8:  var time = LocalTime.of(6, 15);
9:  var dateTime = LocalDateTime.of(date, time);
10: var duration = Duration.ofHours(6);

11: System.out.println(dateTime.plus(duration));  // 2025–01–
20T12:15
12: System.out.println(time.plus(duration));      // 12:15
13: System.out.println(
14:    date.plus(duration));  //
UnsupportedTemporalTypeException
```

Line 11 shows that we can add hours to a `LocalDateTime`, since it contains a time. Line 12 also works, since all we have is a time. Line 13 fails because we cannot add hours to an object that does not contain a time.

Let’s try that again, but add 23 hours this time.

```
7:  var date = LocalDate.of(2025, 1, 20);
8:  var time = LocalTime.of(6, 15);
9:  var dateTime = LocalDateTime.of(date, time);
10: var duration = Duration.ofHours(23);
11: System.out.println(dateTime.plus(duration));  // 2025–01–
21T05:15
12: System.out.println(time.plus(duration));      // 05:15
13: System.out.println(
14:    date.plus(duration));  //
UnsupportedTemporalTypeException
```

This time we see that Java moves forward past the end of the day. Line 11 goes to the next day since we pass midnight. Line 12 doesn’t have a day, so the time just wraps around—just like on a real clock.

Period vs. *Duration*

Remember that `Period` and `Duration` are not equivalent. This example shows a `Period` and `Duration` of the same length:

```
var date = LocalDate.of(2025, 5, 25);
var period = Period.ofDays(1);
var days = Duration.ofDays(1);

System.out.println(date.plus(period));   // 2025–05–26
System.out.println(date.plus(days));     // Unsupported unit:
Seconds
```

Since we are working with a `LocalDate`, we are required to use `Period`. `Duration` has time units in it, even if we don't see them, and they are meant only for objects with time. Make sure you can fill in Table 4.8 to identify which objects can use `Period` and `Duration`.

TABLE 4.8 Where to use `Duration` and `Period`

	Can use with `Period`?	**Can use with `Duration`?**
`LocalDate`	Yes	No
`LocalDateTime`	Yes	Yes
`LocalTime`	No	Yes
`ZonedDateTime`	Yes	Yes

Working with Instants

The `Instant` class represents a specific moment in time in the GMT time zone. Suppose that you want to run a timer.

```
var now = Instant.now();
// Do something time consuming
var later = Instant.now();

var duration = Duration.between(now, later);
System.out.println(duration.toMillis());  // Returns number
milliseconds
```

In our case, the "something time consuming" was just over a second, and the program printed out `1025`.

If you have a `ZonedDateTime`, you can turn it into an `Instant`:

```
var date = LocalDate.of(2025, 5, 25);
var time = LocalTime.of(11, 55, 00);
var zone = ZoneId.of("US/Eastern");
var zonedDateTime = ZonedDateTime.of(date, time, zone);

var instant = zonedDateTime.toInstant(); // 2025-05-
25T15:55:00Z
System.out.println(zonedDateTime); // 2025-05-25T11:55-
04:00[US/Eastern]
System.out.println(instant);       // 2025-05-25T15:55:00Z
```

The last two lines represent the same moment in time. The `ZonedDateTime` includes a time zone. The `Instant` gets rid of the time zone and turns it into an `Instant` of time in GMT.

You cannot convert a `LocalDateTime` to an `Instant`. Remember that an `Instant` is a point in time. A `LocalDateTime` does not contain a time zone, and it is therefore not universally recognized around the world as the same moment in time.

Accounting for Daylight Saving Time

Some countries observe *daylight saving time*. This is where the clocks are adjusted by an hour twice a year to make better use of the sunlight. Not all countries participate, and those that do use different weekends for the change. You only have to work with U.S. daylight saving time on the exam, and that's what we describe here.

The exam question will let you know if a date/time mentioned falls on a weekend when the clocks are scheduled to be changed. If it is not mentioned in a question, you can assume that it is a normal weekend. The act of moving the clock forward or back occurs at 2:00 a.m., which falls very early Sunday morning.

Figure 4.10 shows what happens with the clocks. When we change our clocks in March, time springs forward from 1:59 a.m. to 3:00 a.m. When we change our clocks in November, time falls back, and we experience the hour from 1:00 a.m. to 1:59 a.m. twice. Children learn this as "Spring forward in the spring, and fall back in the fall."

An image illsutrates the changeover clock timings. On normal days, the time changes happens 1.00 a.m - 1.59 a.m to 2.00 a.m -3.00 a.m and then to 3.00 a.m - 4.00 a.m. On march, the time changes from 1.00 a.m - 1.59 a.m to 3.00 a.m - 4.00 a.m. On november, the time changes from 1.00 a.m - 1.59 a.m in first time to 1.00 a.m - 1.59 a.m again to 2.00 a.m - 4.00 a.m.

FIGURE 4.10 How daylight saving time works

For example, on March 9, 2025, we move our clocks forward an hour and jump from 2:00 a.m. to 3:00 a.m. This means that there is no 2:30 a.m. that day. If we wanted to know the time an hour later than 1:30, it would be 3:30.

```
var date = LocalDate.of(2025, Month.MARCH, 9);
var time = LocalTime.of(1, 30);
var zone = ZoneId.of("US/Eastern");
var dateTime = ZonedDateTime.of(date, time, zone);

System.out.println(dateTime);  // 2025–03-09T01:30-
05:00[US/Eastern]
System.out.println(dateTime.getHour());   // 1
System.out.println(dateTime.getOffset()); // -05:00

dateTime = dateTime.plusHours(1);
System.out.println(dateTime);  // 2025–03-09T03:30-
04:00[US/Eastern]
System.out.println(dateTime.getHour());   // 3
System.out.println(dateTime.getOffset()); // -04:00
```

Notice that two things change in this example. The time jumps from 1:30 to 3:30. The UTC offset also changes. Remember when we calculated GMT time by subtracting the time zone from the time? You can see that we went from 6:30 GMT (1:30 minus –5:00) to 7:30 GMT (3:30 minus –4:00). This shows that the time really did change by one hour from GMT's point of view. We printed the hour and offset fields separately for emphasis.

Similarly, in November, an hour after the initial 1:30 a.m. is also 1:30 a.m. because at 2:00 a.m. we repeat the hour. This time, try to calculate the GMT time yourself for all three times to confirm that we really do move only one hour at a time.

```
var date = LocalDate.of(2025, Month.NOVEMBER, 2);
var time = LocalTime.of(1, 30);
var zone = ZoneId.of("US/Eastern");
var dateTime = ZonedDateTime.of(date, time, zone);
System.out.println(dateTime); // 2025-11-02T01:30-
04:00[US/Eastern]

dateTime = dateTime.plusHours(1);
System.out.println(dateTime); // 2025-11-02T01:30-
05:00[US/Eastern]

dateTime = dateTime.plusHours(1);
System.out.println(dateTime); // 2025-11-02T02:30-
05:00[US/Eastern]
```

Did you get it? We went from 5:30 GMT to 6:30 GMT, to 7:30 GMT.

Finally, trying to create a time that doesn’t exist just rolls forward:

```
var date = LocalDate.of(2025, Month.MARCH, 9);
var time = LocalTime.of(2, 30);
var zone = ZoneId.of("US/Eastern");
var dateTime = ZonedDateTime.of(date, time, zone);
System.out.println(dateTime);    // 2025–03–09T03:30–
04:00[US/Eastern]
```

Java is smart enough to know that there is no 2:30 a.m. that night and switches over to the appropriate GMT offset.

Yes, it is annoying that Oracle expects you to know this even if you aren’t in the United States—or for that matter, in a part of the United States that doesn’t follow daylight saving time. The exam creators are in the United States, and they decided that everyone needs to know how U.S. time zones work.

Summary

In this chapter, you learned that a `String` is an immutable sequence of characters. Calling the constructor explicitly 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.

By contrast, a `StringBuilder` is a mutable sequence 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 an array of arrays, the second-level arrays and beyond can be different sizes.

The `Math` class provides a number of `static` methods for performing mathematical operations. For example, you can get minimums or maximums. You can round or even generate random numbers. Some methods work on any numeric primitive, and others only work on `double`.

A `LocalDate` contains just a date, a `LocalTime` contains just a time, and a `LocalDateTime` contains both a date and a time. All three have private constructors and are created using `LocalDate.now()` or `LocalDate.of()` (or the equivalents for that class). Dates and times can be manipulated using `plus ()`, `minus ()`, `at()`, or `with()` methods. The `Period` class represents a number of days, months, or years to add to or subtract from a `LocalDate` or `LocalDateTime`. The date and time classes are all immutable, which means the return value must be used.

Exam Essentials

Be able to determine the output of code using *String*. Know the rules for concatenating with `String` and how to use common `String` methods. Know that a `String` is immutable. Pay special attention to the fact that indexes are zero-based and that the `substring()` method 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 a `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 the `String` class, `equals()` checks the characters inside of it.

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

Identify the return types of *Math* methods. Depending on the primitive passed in, the `Math` methods may return different primitive results.

Recognize invalid uses of dates and times. `LocalDate` does not contain time fields, and `LocalTime` does not contain date fields. Watch for operations being performed on the wrong type. Also watch for adding or subtracting time and ignoring the result. Be comfortable with date math, including time zones and daylight saving time.

Review Questions

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

1. What is output by the following code?

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

   A. `4 1`

   B. `5`

   C. `5 tuna`

   D. `5tuna`