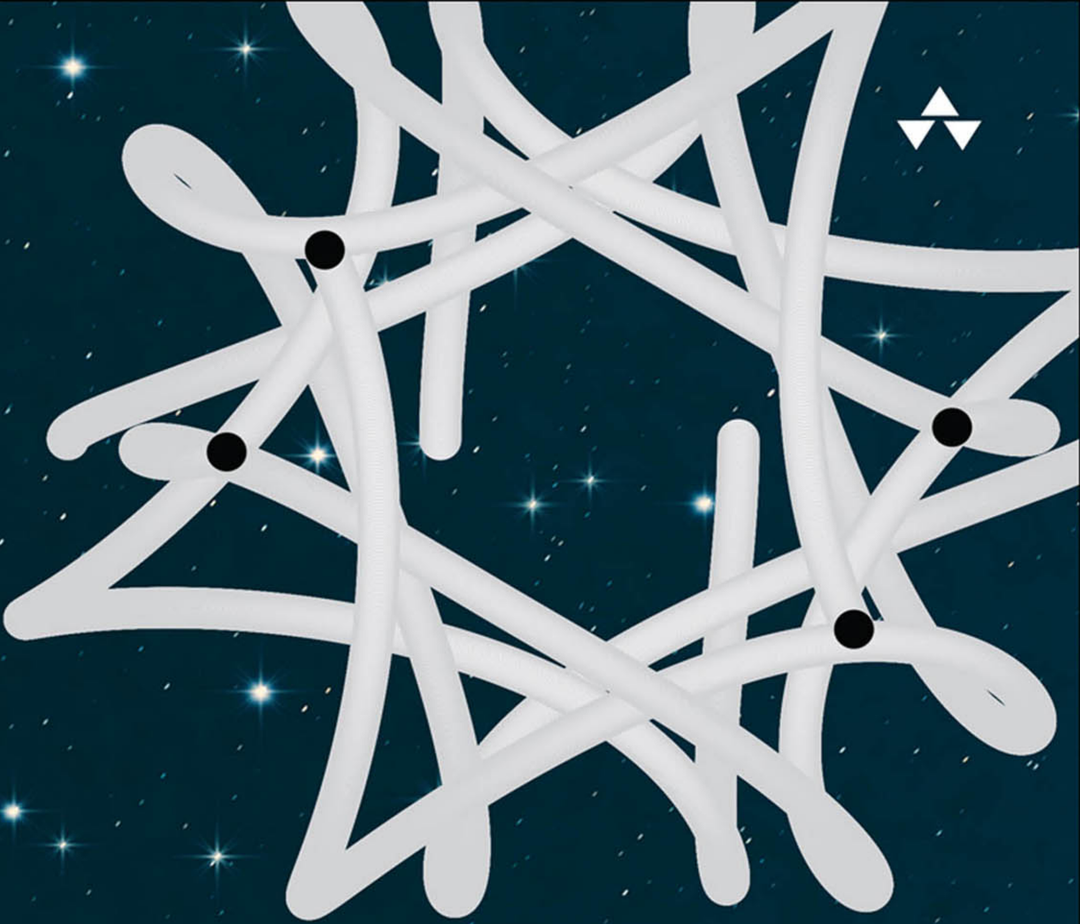


Computer Science

An Interdisciplinary Approach

ROBERT SEDGEWICK
KEVIN WAYNE

Computer
Science

3.1 Using Data Types

ORGANIZING DATA FOR PROCESSING IS AN essential step in the development of a computer program. Programming in Java is largely based on doing so with data types known as *reference types* that are designed to support object-oriented programming, a style of programming that facilitates organizing and processing data.

The eight primitive data types (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`) that you have been using are supplemented in Java by extensive libraries of reference types that are tailored for a large variety of applications. The `String` data type is one such example that you have already used. You will learn more about the `String` data type in this section, as well as how to use several other reference types for image processing and input/output. Some of them are built into Java (`String` and `Color`), and some were developed for this book (`In`, `Out`, `Draw`, and `Picture`) and are useful as general resources.

3.1.1	Identifying a potential gene	337
3.1.2	Albers squares	342
3.1.3	Luminance library	345
3.1.4	Converting color to grayscale	348
3.1.5	Image scaling	350
3.1.6	Fade effect	352
3.1.7	Concatenating files	356
3.1.8	Screen scraping for stock quotes	359
3.1.9	Splitting a file	360

Programs in this section

In the first two chapters of this book, most of our programs were confined to operations on numbers. Of course, the reason is that all of Java's primitive types represent numbers (or truth values). The one exception has been strings, a reference type that is built into Java. With reference types you can write programs that operate not just on strings, but on images, sounds, or any of hundreds of other abstractions that are available in Java's libraries or on our booksite.

In this section, we focus on client programs that *use* existing data types, to give you some concrete reference points for understanding these new concepts and to illustrate their broad reach. We will consider programs that manipulate strings, colors, images, files, and web pages—quite a leap from the primitive types of CHAPTER 1.

In the next section, you will take another leap, by learning how to *define* your own data types to implement any abstraction whatsoever, taking you to a whole new level of programming. Writing programs that operate on your own types of data is an extremely powerful and useful style of programming that has dominated the landscape for many years.

Basic definitions A *data type is a set of values and a set of operations defined on those values*. This statement is one of several mantras that we repeat often because of its importance. In Chapter 1, we discussed in detail Java's *primitive* data types. For example, the values of the primitive data type `int` are integers between -2^{31} and $2^{31} - 1$; the operations defined for the `int` data type include those for basic arithmetic and comparisons, such as `+`, `*`, `%`, `<`, and `>`.

You also have been using a data type that is not primitive—the `String` data type. You know that values of the `String` data type are sequences of characters and that you can perform the operation of concatenating two `String` values to produce a `String` result. You will learn in this section that there are dozens of other operations available for processing strings, such as finding a string's length, extracting individual characters from the string, and comparing two strings.

Every data type is defined by its set of values and the operations defined on them, but when we *use* the data type, we focus on the *operations*, not the values. When you write programs that use `int` or `double` values, you are not concerning yourself with *how* they are represented (we never did spell out the details), and the same holds true when you write programs that use reference types, such as `String`, `Color`, or `Picture`. In other words, *you do not need to know how a data type is implemented to be able to use it* (yet another mantra)

The `String` data type. As a running example, we will revisit Java's `String` data type in the context of object-oriented programming. We do so for two reasons. First, you have been using the `String` data type since your first program, so it is a familiar example. Second, string processing is critical to many computational applications. Strings lie at the heart of our ability to compile and run Java programs and to perform many other core computations; they are the basis of the information-processing systems that are critical to most business systems; people use them every day when typing into email, blog, or chat applications or preparing documents for publication; and they have proved to be critical ingredients in scientific progress in several fields, particularly molecular biology.

We will write programs that declare, create, and manipulate values of type `String`. We begin by describing the `String` API, which documents the available operations. Then, we consider Java language mechanisms for *declaring variables*, *creating objects* to hold data-type values, and *invoking instance methods* to apply data-type operations. These mechanisms differ from the corresponding ones for primitive types, though you will notice many similarities.

API. The Java *class* provides a mechanism for defining data types. In a class, we specify the data-type values and implement the data-type operations. To fulfill our promise that *you do not need to know how a data type is implemented to be able to use it*, we specify the behavior of classes for clients by listing their instance methods in an *API* (*application programming interface*), in the same manner as we have been doing for libraries of static methods. The purpose of an API is to provide the information that you need to write a client program that uses the data type.

The following table summarizes the instance methods from Java's `String` API that we use most often; the full API has more than 60 methods! Several of the methods use integers to refer to a character's index within a string; as with arrays, these indices start at 0.

`public class String` *(Java string data type)*

	`String(String s)`	*create a string with the same value as* `s`
	`String(char[] a)`	*create a string that represents the same sequence of characters as in* `a[]`
`int`	`length()`	*number of characters*
`char`	`charAt(int i)`	*the character at index* `i`
`String`	`substring(int i, int j)`	*characters at indices* `i` *through* (`j-1`)
`boolean`	`contains(String substring)`	*does this string contain* `substring`*?*
`boolean`	`startsWith(String prefix)`	*does this string start with* `prefix`*?*
`boolean`	`endsWith(String postfix)`	*does this string end with* `postfix`*?*
`int`	`indexOf(String pattern)`	*index of first occurrence of* `pattern`
`int`	`indexOf(String pattern, int i)`	*index of first occurrence of* `pattern` *after* `i`
`String`	`concat(String t)`	*this string, with* `t` *appended*
`int`	`compareTo(String t)`	*string comparison*
`String`	`toLowerCase()`	*this string, with lowercase letters*
`String`	`toUpperCase()`	*this string, with uppercase letters*
`String`	`replace(String a, String b)`	*this string, with* `a`*s replaced by* `b`*s*
`String`	`trim()`	*this string, with leading and trailing whitespace removed*
`String[]`	`split(String delimiter)`	*strings between occurrences of* `delimiter`
`boolean`	`equals(Object t)`	*is this string's value the same as* `t`*'s?*
`int`	`hashCode()`	*an integer hash code*

See the online documentation and booksite for many other available methods.

Excerpts from the API for Java's `String` *data type*

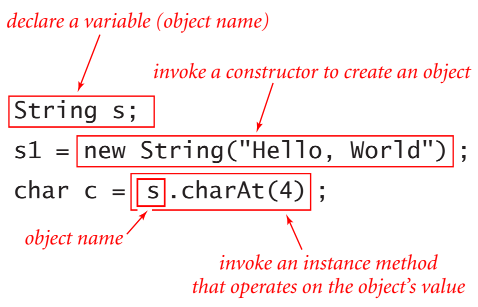
The first entry, with the same name as the class and no return type, defines a special method known as a *constructor*. The other entries define *instance methods* that can take arguments and return values in the same manner as the static methods that we have been using, but they are *not* static methods: they implement operations for the data type. For example, the instance method `length()` returns the number of characters in the string and `charAt()` returns the character at a specified index.

Declaring variables. You declare variables of a reference type in precisely the same way that you declare variables of a primitive type, using a declaration statement consisting of the data type name followed by a variable name. For example, the statement

```
String s;
```

declares a variable `s` of type `String`. This statement does not *create* anything; it just says that we will use the variable name `s` to refer to a `String` object. By convention, reference types begin with uppercase letters and primitive types begin with lowercase letters.

Creating objects. In Java, each data-type value is stored in an *object*. When a client invokes a constructor, the Java system creates (or *instantiates*) an individual object (or *instance*). To invoke a constructor, use the keyword `new`; followed by the class name; followed by the constructor's arguments, enclosed in parentheses and separated by commas, in the same manner as a static method call. For example, `new String("Hello, World")` creates a new `String` object corresponding to the sequence of characters `Hello, World`. Typically, client code invokes a constructor to create an object and assigns it to a variable in the same line of code as the declaration:


```
String s;
s1 = new String("Hello, World");
char c = s.charAt(4);
```
declare a variable (object name)
invoke a constructor to create an object
object name
invoke an instance method that operates on the object's value
Using a reference data type

```
String s = new String("Hello, World");
```

You can create any number of objects from the same class; each object has its own identity and may or may not store the same value as another object of the same type. For example, the code

```
String s1 = new String("Cat");
String s2 = new String("Dog");
String s3 = new String("Cat");
```

creates three different `String` objects. In particular, `s1` and `s3` refer to different objects, even though the two objects represent the same sequence of characters.

Invoking instance methods. The most important difference between a variable of a reference type and a variable of a primitive type is that you can use reference-type variables to invoke the methods that implement data-type operations (in contrast to the built-in syntax involving operators such as + and * that we used with primitive types). Such methods are known as *instance methods*. *Invoking* (or *calling*) an instance method is similar to calling a static method in another class, except that an instance method is associated not just with a class, but also with an individual object. Accordingly, we typically use an *object* name (variable of the given type) instead of the *class* name to identify the method. For example, if `s1` and `s2` are variables of type `String` as defined earlier, then `s1.length()` returns the integer 3, `s1.charAt(1)` returns the character `'a'`, and `s1.concat(s2)` returns a new string `CatDog`.

```
String a = new String("now is");
String b = new String("the time");
String c = new String(" the");
```

instance method call	*return type*	*return value*
`a.length()`	`int`	`6`
`a.charAt(4)`	`char`	`'i'`
`a.substring(2, 5)`	`String`	`"w i"`
`b.startsWith("the")`	`boolean`	`true`
`a.indexOf("is")`	`int`	`4`
`a.concat(c)`	`String`	`"now is the"`
`b.replace("t", "T")`	`String`	`"The Time"`
`a.split(" ")`	`String[]`	`{ "now", "is" }`
`b.equals(c)`	`boolean`	`false`

Examples of `String` data-type operations

String shortcuts. As you already know, Java provides special language support for the `String` data type. You can create a `String` object using a string literal instead of an explicit constructor call. Also, you can concatenate two strings using the string concatenation operator (+) instead of making an explicit call to the `concat()` method. We introduced the longhand version here solely to demonstrate the syntax you need for other data types; these two shortcuts are unique to the `String` data type.

shorthand	`String s = "abc";`	`String t = r + s;`
longhand	`String s = new String("abc");`	`String t = r.concat(s);`

The following code fragments illustrate the use of various string-processing methods. This code clearly exhibits the idea of developing an abstract model and separating the code that implements the abstraction from the code that uses it. This ability characterizes object-oriented programming and is a turning point in this book: we have not yet seen any code of this nature, but virtually all of the code that we write from this point forward will be based on defining and invoking methods that implement data-type operations.

extract file name and extension from a command-line argument

```
String s = args[0];
int dot = s.indexOf(".");
String base      = s.substring(0, dot);
String extension = s.substring(dot + 1, s.length());
```

print all lines on standard input that contain a string specified as a command-line argument

```
String query = args[0];
while (StdIn.hasNextLine())
{
   String line = StdIn.readLine();
   if (line.contains(query))
      StdOut.println(line);
}
```

is the string a palindrome?

```
public static boolean isPalindrome(String s)
{
   int n = s.length();
   for (int i = 0; i < n/2; i++)
      if (s.charAt(i) != s.charAt(n-1-i))
         return false;
   return true;
}
```

translate from DNA to mRNA (replace `'T'` *with* `'U'`*)*

```
public static String translate(String dna)
{
   dna = dna.toUpperCase();
   String rna = dna.replace("T", "U");
   return rna;
}
```

Typical string-processing code