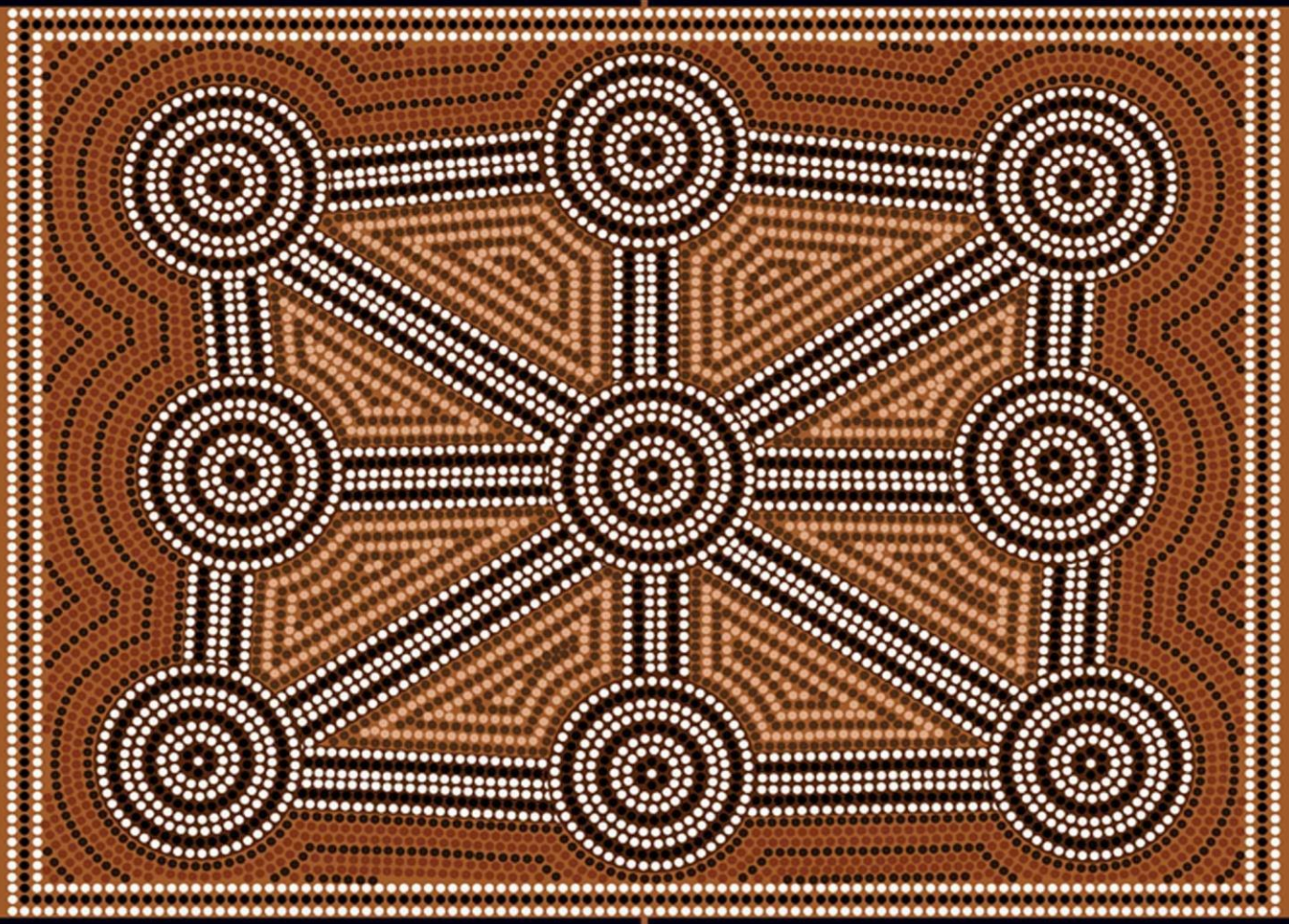
SIXTH EDITION

DATA STRUCTURES & ALGORITHMS

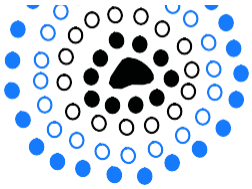


MICHAEL T. GOODRICH
ROBERTO TAMASSIA
MICHAEL H. GOLDWASSER

in JAVA™

WILEY

Chapter 1 Java Primer



Contents

1.1 Strings, Wrappers, Arrays, and Enum Types

The String Class

Java's `char` base type stores a value that represents a single text ***character***. In Java, the set of all possible characters, known as an ***alphabet***, is the Unicode international character set, a 16-bit character encoding that covers most used written languages. (Some programming languages use the smaller ASCII character set, which is a proper subset of the Unicode alphabet based on a 7-bit encoding.) The form for expressing a character literal in Java is using single quotes, such as `'G'`.

```
String title = "Data Structures & Algorithms in Java"
```

Character Indexing

Each character c within a string s can be referenced by using an ***index***, which is equal to the number of characters that come before c in s. By this convention, the first character is at index 0, and the last is at index $n - 1$, where n is the length of the string. For example, the string `title`, defined above, has length 36. The character at index 2 is `'t'` (the third character), and the character at index 4 is `' '` (the space character). Java's `String` class supports a method `length()`, which returns the length of a string instance, and a method `charAt(`k`)`, which returns the character at index k.

Concatenation

The primary operation for combining strings is called ***concatenation***, which takes a string P and a string Q combines them into a new string, denoted $P + Q$, which consists of all the characters of P followed by all the characters of Q. In Java, the "+" operation performs concatenation when acting on two strings, as follows:

```
String term = "over" + "load";
```

The StringBuilder Class

An important trait of Java's `String` class is that its instances are ***immutable***; once an instance is created and initialized, the value of that instance cannot be changed. This is an intentional design, as it allows for great efficiencies and optimizations within the Java Virtual Machine.

```
String greeting = "Hello";
greeting = "Ciao";                    // we changed our mind
```

It is also quite common in Java to use string concatenation to build a new string that is subsequently used to replace one of the operands of concatenation, as in:

```
greeting = greeting + '!';            // now it is "Ciao!"
```

For long string (such as DNA sequences), this can be very time consuming.

`setCharAt(`*k*`, `*c*`)`: Change the character at index *k* to character *c*.

`insert(`*k*`, `*s*`)`: Insert a copy of string *s* starting at index *k* of the sequence, shifting existing characters further back to make room.

`append(`*s*`)`: Append string *s* to the end of the sequence.

`reverse()`: Reverse the current sequence.

`toString()`: Return a traditional `String` instance based on the current character sequence.

Wrapper Types

Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

Table 1.2: Java's wrapper classes. Each class is given with its corresponding base type and example expressions for creating and accessing such objects. For each row, we assume the variable obj is declared with the corresponding class name.

Automatic Boxing and Unboxing

Java provides additional support for implicitly converting between base types and their wrapper types through a process known as automatic boxing and unboxing.

```
1 int j = 8;
2 Integer a = new Integer(12);
3 int k = a;                          // implicit call to a.intValue()
4 int m = j + a;                      // a is automatically unboxed before the addition
5 a = 3 * m;                          // result is automatically boxed before assignment
6 Integer b = new Integer("-135");    // constructor accepts a String
7 int n = Integer.parseInt("2013");   // using static method of Integer class
```

Code Fragment 1.4: A demonstration of the use of the Integer wrapper class.

Arrays

A common programming task is to keep track of an ordered sequence of related values or objects. For example, we may want a video game to keep track of the top ten scores for that game. Rather than using ten different variables for this task, we would prefer to use a single name for the group and use index numbers to refer to the high scores in that group. Similarly, we may want a medical information system to keep track of the patients currently assigned to beds in a certain hospital. Again, we would rather not have to introduce 200 variables in our program just because the hospital has 200 beds.

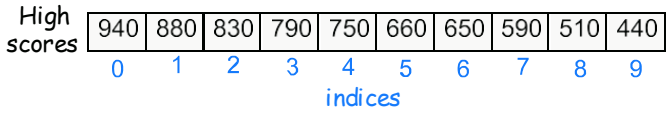


Figure 1.3: An illustration of an array of ten (int) high scores for a video game.

Array Elements and Capacities

Each value stored in an array is called an ***element*** of that array. Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its ***capacity***. In Java, the length of an array named a can be accessed using the syntax a.length. Thus, the cells of an array a are numbered 0, 1, 2, and so on, up through a.length-1, and the cell with index k can be accessed with syntax $a[k]$.

Out of Bounds Errors

It is a dangerous mistake to attempt to index into an array a using a number outside the range from 0 to a.length-1. Such a reference is said to be ***out of bounds***. Out of bounds references have been exploited numerous times by hackers using a method called the ***buffer overflow attack*** to compromise the security of computer systems written in languages other than Java. As a safety feature, array indices are always checked in Java to see if they are ever out of bounds. If an array index is out of bounds, the runtime Java environment signals an error condition. The name of this condition is the ArrayIndexOutOfBoundsException. This check helps Java avoid a number of security problems, such as buffer overflow attacks.

Declaring and Constructing Arrays

Arrays in Java are somewhat unusual, in that they are not technically a base type nor are they instances of a particular class. With that said, an instance of an array is treated as an object by Java, and variables of an array type are *reference variables*.

```
int[ ] primes;
```

Because arrays are a reference type, this declares the variable primes to be a reference to an array of integer values, but it does not immediately construct any such array. There are two ways for creating an array.

elementType[] *arrayName* = {$initialValue_0, initialValue_1, \ldots, initialValue_{N-1}$};

The *elementType* can be any Java base type or class name, and *arrayName* can be any valid Java identifier. The initial values must be of the same type as the array. For example, we could initialize the array of primes to contain the first ten prime numbers as:

```
int[ ] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

When using an initializer, an array is created having precisely the capacity needed to store the indicated values.

new *elementType*[*length*]

where *length* is a positive integer denoting the length of the new array. The new operator returns a reference to the new array, and typically this would be assigned to an array variable. For example, the following statement declares an array variable named measurements, and immediately assigns it a new array of 1000 cells.

```
double[ ] measurements = new double[1000];
```

When arrays are created using the new operator, all of their elements are automatically assigned the default value for the element type. That is, if the element type is numeric, all cells of the array are initialized to zero, if the element type is boolean, all cells are false, and if the element type is a reference type (such as with an array of String instances), all cells are initialized to null.

Enum Types

In olden times, programmers would often define a series of constant integer values to be used for representing a finite set of choices. For example, in representing a day of the week, they might declare variable today as an int and then set it with value 0 for Monday, 1 for Tuesday, and so on.

A slightly better programming style is to define static constants (with the final keyword), to make the associations, such as:

```
static final int MON = 0;
static final int TUE = 1;
static final int WED = 2;
...
```

Java supports a more elegant approach to representing choices from a finite set by defining what is known as an enumerated type, or enum for short. These are types that are only allowed to take on values that come from a specified set of names. They are declared as follows:

modifier enum *name* { $valueName_0$, $valueName_1$, ..., $valueName_{n-1}$ };

where the *modifier* can be blank, public, protected, or private. The name of this enum, *name*, can be any legal Java identifier. Each of the value identifiers, $valueName_i$, is the name of a possible value that variables of this enum type can take on. Each of these name values can also be any legal Java identifier, but the Java convention is that these should usually be capitalized words. For example, an enumerated type definition for days of the weak might appear as:

```
public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
```

Once defined, Day becomes an official type and we may declare variables or parameters with type Day. A variable of that type can be declared as:

```
Day today;
```

and an assignment of a value to that variable can appear as:

```
today = Day.TUE;
```