

String, StringBuilder, Arrays, and ArrayList

Exam objectives covered in this chapter	What you need to know
[2.7] Create and manipulate strings.	How to create <code>String</code> objects using the assignment and <code>new</code> operators. Use of the operators <code>=</code> , <code>+=</code> , <code>!=</code> , and <code>==</code> with <code>String</code> objects. Literal value for class <code>String</code> . Use of methods from class <code>String</code> . Immutable <code>String</code> values. All of the <code>String</code> methods manipulate and return a new <code>String</code> object.
[3.3] Test equality between strings and other objects using <code>==</code> and <code>equals()</code> .	How to determine the equality of two <code>String</code> objects. Differences between using operator <code>==</code> and method <code>equals()</code> to determine equality of <code>String</code> objects.
[2.6] Manipulate data using the <code>StringBuilder</code> class and its methods.	How to create <code>StringBuilder</code> classes and how to use their commonly used methods. Difference between <code>StringBuilder</code> and <code>String</code> classes. Difference between methods with similar names defined in both of these classes.
[4.1] Declare, instantiate, initialize, and use a one-dimensional array.	How to declare, instantiate, and initialize one-dimensional arrays using single and multiple steps. The do's and don'ts of each of these steps.
[4.2] Declare, instantiate, initialize, and use a multidimensional array.	How to declare, instantiate, and initialize multidimensional arrays using single and multiple steps, with do's and don'ts for each of these steps. Accessing elements in asymmetric multidimensional arrays.

Exam objectives covered in this chapter	What you need to know
[4.3] Declare and use an ArrayList.	How to declare, create, and use an ArrayList. Advantages of using an ArrayList over arrays. Use of methods that add, modify, and delete elements of an ArrayList.

In the OCA Java SE 7 Programmer I exam, you'll be asked many questions about how to create, modify, and delete String objects, StringBuilder objects, arrays, and ArrayList objects. To prepare you for such questions, in this chapter I'll provide insight into the variables you'll use to store these objects' values, along with definitions for some of their methods. This information should help you apply all of the methods correctly.

In this chapter, we'll cover the following:

- Creating and manipulating String and StringBuilder objects
- Using common methods from class String and StringBuilder
- Creating and using one-dimensional and multidimensional arrays in single and multiple steps
- Accessing elements in asymmetric multidimensional arrays
- Declaring, creating, and using an ArrayList and understanding the advantages of an ArrayList over arrays
- Using methods that add, modify, and delete elements of an ArrayList

4.1 Welcome to the world of the String class



[2.7] Create and manipulate strings



[3.3] Test equality between strings and other objects using == and equals()

In this section, we'll cover the class `String` defined in the Java API in the `java.lang` package. The `String` class represents character strings. We'll create objects of the class `String` and work with its commonly used methods, including `indexOf()`, `substring()`, `replace()`, `charAt()`, and others. You'll also learn how to determine the equality of two `String` objects.

The `String` class is perhaps the most used class in the Java API. You'll find instances of this class being used by every other class in the Java API. How many times do you think you've used the class `String`? Don't answer that question—it's like trying to count your hair.

Although many developers find the `String` class to be one of the simplest to work with, this perception can be deceptive. For example, in the `String` value "Shreya", at which position do you think is `r` stored—second or third? The correct answer is second because the first letter of a `String` is stored at position 0 and not position 1. You'll learn many other facts about the `String` class in this section.

Let's start by creating new objects of this class.

4.1.1 Creating String objects

You can create objects of the class `String` by using the `new` operator, by using the assignment operator (`=`), or by enclosing a value within double quotes (`"`). But you may have noticed a *big* difference in how these objects are created, stored, and referred by Java.

Let's create two `String` objects with the value "Paul" using the operator `new`:

```
String str1 = new String("Paul");
String str2 = new String("Paul");
System.out.println(str1 == str2);
```

Create two String objects by using the operator new

When comparing the objects referred to by the variables str1 and str2, it prints false.

Figure 4.1 illustrates the previous code.

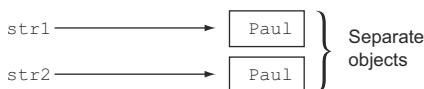


Figure 4.1 String objects created using the operator new always refer to separate objects, even if they store the same sequence of characters.

In the previous code, a comparison of the `String` reference variables `str1` and `str2` prints `false`. The operator `==` compares the addresses of the objects referred to by the variables `str1` and `str2`. Even though these `String` objects store the same sequence of characters, they refer to separate objects that are stored at separate locations.

Let's create two `String` objects with the value "Harry" using the assignment operator (`=`). Figure 4.2 illustrates the variables `str3` and `str4` and the objects referred to by these variables.

```
String str3 = "Harry";
String str4 = "Harry";
System.out.println(str3 == str4);
```

Create two String objects by using assignment operator =

Prints true because str3 and str4 refer to the same object

In the previous example, the variables `str1` and `str2` referred to different `String` objects, even if they were created using the same sequence of characters. In the case of variables `str3` and `str4`, the objects are created and stored in a *pool* of `String` objects. Before creating a new object in the pool, Java first searches for an object with similar contents. When the following line of code executes, no `String` object with the value "Harry" is found in the pool of `String` objects:

```
String str3 = "Harry";
```

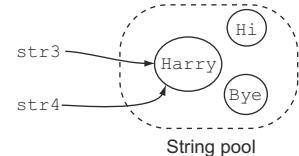


Figure 4.2 String objects created using the assignment operator (=) may refer to the same object if they store the same sequence of characters.

As a result, Java creates a `String` object with the value "Harry" in the pool of `String` objects referred to by variable `str3`. This action is depicted in figure 4.3.

When the following line of code executes, Java is able to find a `String` object with the value "Harry" in the pool of `String` objects:

```
String str4 = "Harry";
```

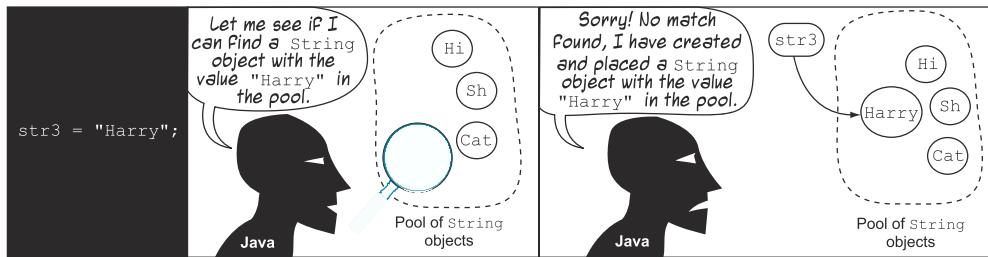


Figure 4.3 The sequence of steps that executes when Java is unable to locate a String in a pool of String objects

Java doesn't create a new String object in this case, and the variable str4 refers to the existing String object "Harry". As shown in figure 4.4, both variables str3 and str4 refer to the same String object in the pool of String objects.

You can also create a String object by enclosing a value within double quotes (""):

`System.out.println("Morning");` Creates a new String object with value Morning in the String constant pool

These values are reused from the String constant pool if a matching value is found. If a matching value isn't found, the JVM creates a String object with the specified value and places it in the String constant pool:

```
String morning1 = "Morning";
System.out.println("Morning" == morning1);
```

Compare the preceding example with the following example, which creates a String object using the operator new and (only) double quotes and then compares their references:

```
String morning2 = new String("Morning");
System.out.println("Morning" == morning2);
```

This String object is not placed in the String constant pool

The preceding code shows that object references of String objects that exist in the String constant pool and object references of String objects that don't exist in

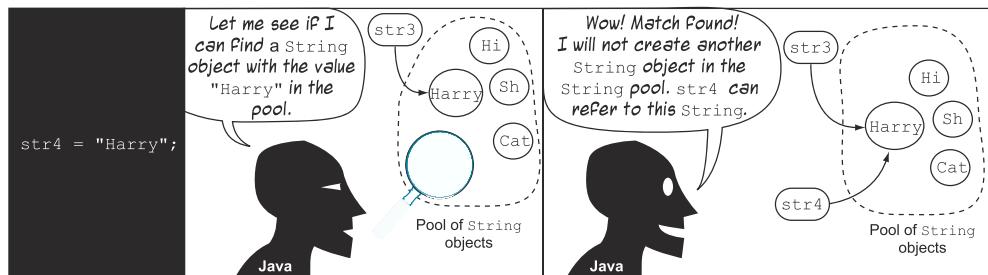


Figure 4.4 The sequence of actions that executes when Java locates a String in the pool of String objects

the String constant pool don't refer to the same String object, even if they define the same String value.



NOTE The terms “String constant pool” and “String pool” are used interchangeably and refer to the same pool of String objects. Because String objects are immutable, the pool of String objects is also called the “String *constant* pool.” You may see either of these terms on the exam.

You can also invoke other overloaded constructors of class String to create its objects by using the operator new:

```
String girl = new String("Shreya");
char[] name = new char[] {'P', 'a', 'u', 'l'};
String boy = new String(name);
```

String constructor
that accepts a String

String constructor that
accepts a char array

You can also create objects of String using the classes StringBuilder and StringBuffer:

```
StringBuilder sd1 = new StringBuilder("String Builder");
String str5 = new String(sd1);
StringBuffer sb2 = new StringBuffer("String Buffer");
String str6 = new String(sb2);
```

String constructor
that accepts object
of StringBuilder

String constructor that
accepts object of StringBuffer

Because String is a class, you can assign null to it, as shown in the next example:

```
String empName = null;
```

null is a literal
value for objects



EXAM TIP The literal value for String is null.

COUNTING STRING OBJECTS

To test your understanding on the various ways in which a String object can be created, the exam may question you on the total number of String objects created in a given piece of code. Count the total number of String objects created in the following code, assuming that the String constant pool doesn't define any matching String values:

```
class ContString {
    public static void main(String... args) {
        String summer = new String("Summer");           ←①
        String summer2 = "Summer";                      ←②
        System.out.println("Summer");
        System.out.println("autumn");                   ←③
        System.out.println("autumn" == "summer");         ←④
        String autumn = new String("Summer");            ←⑤
    }                                              ←⑥
}
```

I'll walk through the code with you step by step to calculate the total number of String objects created:

- The code at ① creates a new String object with the value "Summer". This object is not placed in the String constant pool.

- The code at ② creates a new String object with the value "Summer" and places it in the String constant pool.
- The code at ③ doesn't need to create any new String object. It reuses the String object with the value "Summer" that already existed in the String constant pool.
- The code at ④ creates a new String object with the value "autumn" and places it in the String constant pool.
- The code at ⑤ reuses the String value "autumn" from the String constant pool. It creates a String object with the value "summer" in the String constant pool (note the difference in the case of letters—Java is case sEnSITVe and "Summer" is not same as "summer").
- The code at ⑥ creates a new String object with the value "Summer".
- The previous code creates a total of five String objects.



EXAM TIP If a String object is created using the keyword new, it always results in the creation of a new String object. A new String object gets created using the assignment operator (=) or double quotes only if a matching String object with the same value isn't found in the String constant pool.

4.1.2 *The class String is immutable*

The concept that the class String is immutable is an important point to remember. Once created, the contents of an object of the class String can never be modified. The immutability of String objects helps the JVM reuse String objects, reducing memory overhead and increasing performance.

As shown previously in figure 4.4, the JVM creates a pool of String objects that can be referenced by multiple variables across the JVM. The JVM can make this optimization only because String is immutable. String objects can be shared across multiple reference variables without any fear of changes in their values. If the reference variables str1 and str2 refer to the same String object value "Java", str1 need not worry for its lifetime that the value "Java" might be changed by variable str2.

Let's take a quick look at how the immutability of class String is implemented by the authors of this class:

- The class String stores its values in a private variable of the type char array (char value[]). Arrays are fixed in size and don't grow once initialized.
- This value variable is marked as final in the class String. Note that final is a nonaccess modifier, and a final variable can be initialized only once.
- None of the methods defined in the class String manipulate the individual elements of the array value.

I'll discuss each of these points in detail in the following sections.

Code from Java API classes

To get a better understanding of how classes `String`, `StringBuilder`, and `ArrayList` work, I'll explain the variables used to store these objects' values, along with definitions for some of their methods. My purpose is not to overwhelm you, but to prepare you. The exam won't question you on this subject, but these details will help you retain relevant information for the exam and implement similar requirements in code for practical projects.

The source code of the classes defined in the Java API is shipped with the Java Development Kit (JDK). You can access it by unzipping the folder `src` from your JDK's installation folder.

The rest of this section discusses how the authors of the Java API have implemented immutability in the class `String`.

STRING USES A CHAR ARRAY TO STORE ITS VALUE

Here's a partial definition of the class `String` from the Java source code file (`String.java`) that includes the array used to store the characters of a `String` value (the relevant code is in bold):

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
{
    private final char value[];
    |   The rest of the code
    |   of the class String
    |
    |
}
```

The value array is used
for character storage

The arrays are fixed in size—they can't grow once they're initialized.

Let's create a variable name of type `String` and see how it's stored internally:

```
String name = "Selvan";
```

Figure 4.5 shows a UML representation (class diagram on the left and object diagram on the right) of the class `String` and its object name, with only one relevant variable, `value`, which is an array of the type `char` and is used to store the sequence of characters assigned to a `String`.

As you can see in figure 4.5, the `String` value `Selvan` is stored in an array of type `char`. In this chapter, I'll cover arrays in detail, as well as how an array stores its first value at position 0.

<code>String</code>	<code>name : String</code>
<code>- value : char[]</code>	<code>value = {'S', 'e', 'l', 'v', 'a', 'n'}</code>
<code>:</code>	<code>:</code>

Figure 4.5 UML presentations of the class `String` and a `String` object with only one variable



Figure 4.6 Mapping characters stored by a String with the positions at which they're stored

Figure 4.6 shows how Selvan is stored as a char array.

What do you think you'll get when you request that this String return the character at position 4? If you said a and not v, you got the right answer (as in figure 4.6).

STRING USES FINAL VARIABLE TO STORE ITS VALUE

The variable value, which is used to store the value of a String object, is marked as final. Review the following code snippet from the class `String.java`:

```
private final char value[]; ← | value is used for
                                | character storage
```

The basic characteristic of a final variable is that it can initialize a value only once. By marking the variable `value` as final, the class `String` makes sure that it can't be reassigned a value.

METHODS OF STRING DON'T MODIFY THE CHAR ARRAY

Although we can't reassign a value to a final char array (as mentioned in the previous section), we can reassign its individual characters. Wow—does this mean that the statement “Strings are immutable” isn't completely true?

No, that statement is still true. The char array used by the class `String` is marked private, which means that it isn't accessible outside the class for modification. The class `String` itself doesn't modify the value of this variable, either.

All the methods defined in the class `String`, such as `substring`, `concat`, `toLowerCase`, `toUpperCase`, `trim`, and so on, which seem to modify the contents of the `String` object on which they're called, create and return a new `String` object, rather than modifying the existing value. Figure 4.7 illustrates the partial definition of `String`'s `replace` method.

I'll reiterate that the previous code from the class `String` will help you relate the theory to the code and understand how and why a particular concept works. If you understand a particular concept well in terms of how and why it works, you'll be able to retain that information longer.

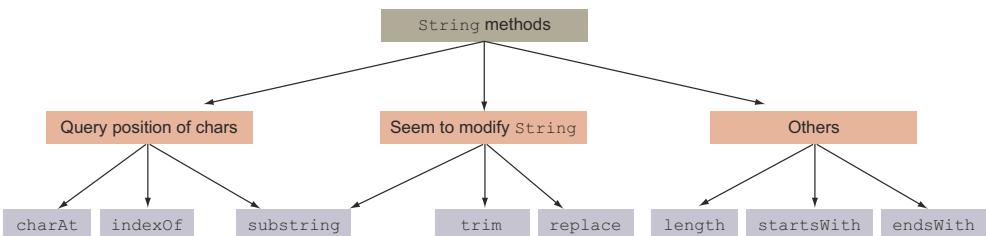
```
public String replace(char oldChar, char newChar) {
    if (oldChar != newChar) {
        // code to create a new char array and
        // replace the desired char with the new char

        return new String(0, len, buf);
    }
    return this;
}
```

↑

replace creates and returns a new String object. It doesn't modify the existing array value.

Figure 4.7 The partial definition of the method replace from the class String shows that this method creates and returns a new String object rather than modifying the value of the String object on which it's called.

**Figure 4.8 Categorization of the String methods**

EXAM TIP Strings are immutable. Once initialized, a String value can't be modified. All the String methods that return a modified String value return a new String object with the modified value. The original String value always remains the same.

4.1.3 Methods of the class String

Figure 4.8 categorizes the methods that are on the exam into groups: ones that query positions of characters, ones that seem to modify String, and others.

Categorizing the methods in this way will help you better understand these methods. For example, the methods `charAt()`, `indexOf()`, and `substring()` query the position of individual characters in a String. The methods `substring()`, `trim()`, and `replace()` seem to be modifying the value of a String.

CHARAT()

You can use the method `charAt(int index)` to retrieve a character at a specified index of a String:

```
String name = new String("Paul");
System.out.println(name.charAt(0));
System.out.println(name.charAt(2));
```

Prints P
Prints u

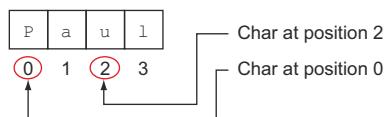
Figure 4.9 illustrates the previous string, Paul.

Because the last character is placed at index 3, the following code will throw an exception at runtime:

```
System.out.println(name.charAt(4));
```



NOTE As a quick introduction, a *runtime exception* is a programming error determined by the Java Runtime Environment (JRE) during the execution of code. These errors occur because of the inappropriate use of another piece of code (exceptions are covered in detail in chapter 7). The previous code tries to access a nonexistent index position, so it causes an exception.

**Figure 4.9 The sequence of characters of "Paul" stored by String and the corresponding array index positions**

INDEXOF()

You can search a String for the occurrence of a char or a String. If the specified char or String is found in the target String, this method returns the first matching position; otherwise, it returns -1:

```
String letters = "ABCAB";
System.out.println(letters.indexOf('B'));
System.out.println(letters.indexOf("S"));
System.out.println(letters.indexOf("CA"));
```

Figure 4.10 illustrates the previous string ABCAB.

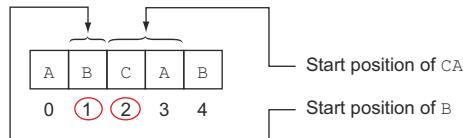


Figure 4.10 The characters "ABCAB" stored by String

By default, the `indexOf()` method starts its search from the first char of the target String. If you wish, you can also set the starting position, as in the following example:

```
String letters = "ABCAB";
System.out.println(letters.indexOf('B', 2));
```

SUBSTRING()

The `substring()` method is shipped in two flavors. The first returns a substring of a String from the position you specify to the end of the String, as in the following example:

```
String exam = "Oracle";
String sub = exam.substring(2);
System.out.println(sub);
```

Figure 4.11 illustrates the previous example.

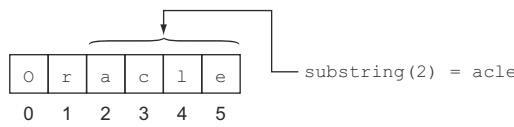


Figure 4.11 The String "Oracle"

You can also specify the end position with this method:

```
String exam = "Oracle";
String result = exam.substring(2, 4);
System.out.println(result);
```

Figure 4.12 illustrates the String value "Oracle", including both a start and end point for the method `substring`.

An interesting point is that the `substring` method doesn't include the character at the end position. In the previous example, `result` is assigned the value `ac` (characters at positions 2 and 3), not the value `acl` (characters at positions 2, 3, and 4).

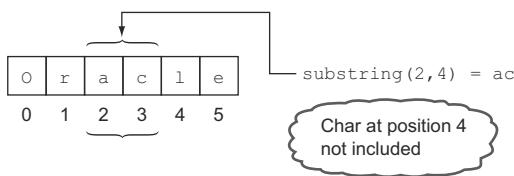


Figure 4.12 How the method `substring` looks for the specified characters from the start until the end position



EXAM TIP The `substring` method doesn't include the character at the end position in its return value.

TRIM()

The `trim()` method returns a new `String` by removing all the leading and trailing *white space* in a `String`. White spaces are blanks (new lines, spaces, or tabs).

Let's define and print a `String` with leading and trailing white space. (The colons printed before and after the `String` determine the start and end of the `String`.)

```
String varWithSpaces = " AB CB      ";
System.out.print(":");
System.out.print(varWithSpaces);
System.out.print(":");
```

← String with white space
← Prints :AB CB : ← Prints :AB CB :

Here's another example that trims the leading and trailing white space:

```
System.out.print(":");
System.out.print(varWithSpaces.trim());
System.out.print(":");
```

← Prints :AB CB: ← Prints :AB CB:

Note that this method doesn't remove the space *within* a `String`.

REPLACE()

This method will return a new `String` by replacing all the occurrences of a `char` with another `char`. Instead of specifying a `char` to be replaced by another `char`, you can also specify a sequence of characters—a `String` to be replaced by another `String`:

```
String letters = "ABCAB";
System.out.println(letters.replace('B', 'b'));
System.out.println(letters.replace("CA", "12"));
```

← Prints AbCAb ← Prints ABI2B

Notice the type of the method parameters passed on this method: either `char` or `String`. You can't mix these parameter types, as the following code shows:

```
String letters = "ABCAB";
System.out.println(letters.replace('B', "b"));
System.out.println(letters.replace("B", 'b'));
```

← Won't compile

Again, notice that this method doesn't—or can't—change the value of the variable `letters`. Examine the following line of code and its output:

```
System.out.println(letters);
```

← Prints ABCAB because previous replace() method calls don't affect the char[] array within letters

LENGTH()

You can use the `length()` method to retrieve the length of a String. Here's an example showing its use:

```
System.out.println("Shreya".length());
```

Prints 6



EXAM TIP The length of a String is one number greater than the position that stores its last character. The length of String "Shreya" is 6, but its last character, a, is stored at position 5 because the positions start at 0, not 1.

STARTSWITH() AND ENDSWITH()

The method `startsWith()` determines whether a String starts with a specified prefix, specified as a String. You can also specify whether you wish to search from the start of a String or from a particular position. This method returns `true` if a match is found and `false` otherwise:

```
String letters = "ABCAB";
System.out.println(letters.startsWith("AB"));
System.out.println(letters.startsWith("a"));
System.out.println(letters.startsWith("A", 3));
```

Prints true
Prints false
Prints true

The method `endsWith()` tests whether a String ends with a particular suffix. It returns `true` for a matching value and `false` otherwise:

```
System.out.println(letters.endsWith("CAB"));
System.out.println(letters.endsWith("B"));
System.out.println(letters.endsWith("b"));
```

Prints true
Prints true
Prints false

METHOD CHAINING

It's common practice to use multiple String methods in a single line of code, as follows:

```
String result = "Sunday ".replace(' ', 'Z').trim().concat("M n");
System.out.println(result);
```

Prints SundayZZM n

The methods are evaluated from left to right. The first method to execute in this example is `replace`, not `concat`.

Method chaining is one of the favorite topics of the exam authors. You're sure to encounter a question on method chaining in the OCA Java SE 7 Programmer I exam.



EXAM TIP When chained, the methods are evaluated from left to right.

Note that there's a difference between calling a chain of methods on a String object versus doing the same and then reassigning the return value to the same variable:

```
String day = "SunDday";
day.replace('D', 'Z').substring(3);
System.out.println(day);
```

Calls methods replace and substring on day.

String is immutable—no change in the value variable day. Prints SunDday.

```
day = day.replace('D', 'Z').substring(3);
System.out.println(day);
```

Calls methods replace and substring on day, and assigns the result back to variable day.

Prints ZDay.

Because `String` objects are immutable, their values won't change if you execute methods on them. You can, of course, reassign a value to a reference variable of type `String`. Watch out for related questions in the exam.

Although the next Twist in the Tale exercise may seem simple, with only two lines of code, appearances can be deceptive (answers in the appendix).

Twist in the Tale 4.1

Let's modify some of the code used in the previous section. Execute this code on your system. Which answer correctly shows its output?

```
String letters = "ABCAB";
System.out.println(letters.substring(0, 2).startsWith('A'));

a true
b false
c AB
d ABC
e Compilation error
```

4.1.4 *String objects and operators*

Of all the operators that are on this exam, you can use just a handful with the `String` objects:

- Concatenation: `+` and `+=`
- Equality: `==` and `!=`

In this section, we'll cover the concatenation operators. We'll cover the equality operators in the next section (4.1.5).

Concatenation operators (`+` and `+=`) have a special meaning for `Strings`. The Java language has additional functionality defined for these operators for `String`. You can use the operators `+` and `+=` to concatenate two `String` values. Behind the scenes, string concatenation is implemented by using the `StringBuilder` (covered in the next section) or `StringBuffer` (similar to `StringBuilder`) classes.

But remember that a `String` is immutable. You can't modify the value of any existing object of `String`. The `+` operator enables you to create a new object of class `String` with a value equal to the concatenated values of multiple `Strings`. Examine the following code:

```
String aString = "OCJA" + "Cert" + "Exam";
```

← aString contains
OCJACertExam

Here's another example:

```
int num = 10;
int val = 12;
String aStr = "OCJA";
```

```
String anotherStr = num + val + aStr;
System.out.println(anotherStr);
```

The code above concatenates the values of variables num, val, and aStr. The variable num is 22, val is OCJA, and aStr is not explicitly defined but is implied to be an empty string or a space. The resulting string is 22OCJA. An annotation 'Prints 22OCJA' with an arrow points to the final output.

Why do you think the value of the variable anotherStr is 22OCJA and not 1012OCJA? The + operator can be used with the primitive values, and the expression num + val + aStr is evaluated from left to right. Here's the sequence of steps executed by Java to evaluate the expression:

- Adds operands num and val to get 22.
- Concatenates 22 with OCJA to get 22OCJA.

If you wish to treat the numbers stored in variables num and val as String values, modify the expression as follows:

```
anotherStr = "" + num + val + aStr;
```

The code above uses the empty string "" as the first operand. This forces the addition of num and val to be performed as String concatenation. The resulting string is 1012OCJA. An annotation 'Evaluates to 1012OCJA' with an arrow points to the final output.

A practical tip on String concatenation

During my preparation for my Java Programmer certification, I learned how the output changes in String concatenation when the order of values being concatenated is changed. At work, it helped me to quickly debug a Java application that was logging incorrect values to a log file. It didn't take me long to discover that the offending line of code was `logToFile("Shipped:" + numReceived() + inTransit());`. The methods were returning correct values individually, but the return values of these methods were not being added. They were being concatenated as String values, resulting in the unexpected output.

One solution is to enclose the int addition within parentheses, as in `logToFile("Shipped:" + (numReceived() + inTransit()));`. This code will log the text "Shipped" with the sum of the numeric values returned by the methods numReceived() and inTransit().

When you use += to concatenate String values, ensure that the variable you're using has been initialized (and doesn't contain null). Look at the following code:

```
String lang = "Java";
lang += " is everywhere!";
String initializedToNull = null;
initializedToNull += "Java";
System.out.println(initializedToNull);
```

The code above shows a common mistake. The variable lang is initialized to "Java" and then concatenated with " is everywhere!". The resulting string is "java is everywhere". However, the variable initializedToNull is initialized to null and then concatenated with "Java". Since initializedToNull is null, the resulting string is nullJava. Annotations indicate that lang is assigned "java is everywhere" and the final output is Prints nullJava.

4.1.5 Determining equality of Strings

The correct way to compare two String values for equality is to use the equals method defined in the String class. This method returns a true value if the object being compared to it isn't null, is a String object, and represents the same sequence of characters as the object to which it's being compared.

The following listing shows the method definitions of the equals method defined in class String in the Java API.

Listing 4.1 Method definition of the equals method from the class String

```

public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = count;
        if (n == anotherString.count) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = offset;
            int j = anotherString.offset;
            while (n-- != 0) {
                if (v1[i++] != v2[j++])
                    return false;
            }
            return true;
        }
    }
    return false;
}

```

The diagram illustrates the logic of the `equals` method with numbered annotations:

- 1** Returns true if the object being compared is the same object.
- 2** Executes statements in if construct if `anObject` is of type `String`.
- 3** Continues comparison if the length of `String` values being compared is equal.
- 4** Compares individual `String` characters; returns false if there's a mismatch with any individual `String` character.
- 5** Returns true if all characters of `String` `anObject` successfully matched with this object.
- 6** Returns false if the object being compared to is not of type `String` or the lengths of the comparing `Strings` don't match.

In listing 4.1, the `equals` method accepts a method parameter of type `Object` and returns a boolean value. Let's walk through the `equals` method defined by class `String`:

- **1** compares the object reference variables. If the reference variables are the same, they refer to the same equal object.
- **2** compares the type of the method parameter to this object. If the method parameter passed to this method is not of type `String`, **6** returns false.
- **3** checks whether the lengths of the `String` values being compared are equal.
- **4** compares the individual characters of the `String` values. It returns false if a mismatch is found at any position. If no mismatch is found, **5** returns true.

Examine the following code:

```

String var1 = new String("Java");
String var2 = new String("Java");
System.out.println(var1.equals(var2));           ← Prints true
System.out.println(var1 == var2);                ← Prints false

```

The operator `==` compares the reference variables, that is, whether the variables refer to the same object. Hence, `var1 == var2` in the previous code prints false. Now examine the following code:

```

String var3 = "code";
String var4 = "code";
System.out.println(var3.equals(var4));           ← Prints true
System.out.println(var3 == var4);                ← Prints true

```

Even though comparing `var3` and `var4` using the operator `==` prints true, you should *never* use this operator for comparing `String` values. The variables `var3` and `var4` refer

to the same `String` object created and shared in the pool of `String` objects. (We discussed the pool of `String` objects in section 4.1.1 earlier in this chapter). This operator won't always return the value `true`, even if the two objects store the same `String` values.



EXAM TIP The operator `==` compares whether the reference variables refer to the same objects, and the method `equals` compares the `String` values for equality. Always use the `equals` method to compare two `String`s for equality. Never use the `==` operator for this purpose.

You can use the operator `!=` to compare the inequality of objects referred to by the `String` variables. It's the inverse of the operator `==`. Let's compare the usage of the operator `!=` with the operator `==` and the method `equals()`:

```
String var1 = new String("Java");
String var2 = new String("Java");
System.out.println(var1.equals(var2));
System.out.println(var1 == var2);
System.out.println(var1 != var2);
```

The following example uses the operators `!=` and `==` and the method `equals` to compare `String` variables that refer to the same object in the `String` constant pool:

```
String var3 = "code";
String var4 = "code";
System.out.println(var3.equals(var4));
System.out.println(var3 == var4);
System.out.println(var3 != var4);
```

As you can see, in both of the previous examples the operator `!=` returns the inverse of the value returned by the operator `==`.

Because `Strings` are immutable, we also need a mutable sequence of characters that can be manipulated. Let's take a look at the other type of `String` on the OCA Java SE Programmer I exam: `StringBuilder`.

4.2 **Mutable strings: *StringBuilder***



[2.6] Manipulate data using the `StringBuilder` class and its methods

The class `StringBuilder` is defined in the package `java.lang` and it has a mutable sequence of characters. You must use class `StringBuilder` when you're dealing with larger strings or modifying the contents of a string often. Doing so will improve the performance of your code. Unlike `StringBuilder`, the `String` class has an immutable sequence of characters. Every time you modify a string that's represented by the `String` class, your code actually creates new `String` objects instead of modifying the existing one.



EXAM TIP You can expect questions on the need for the `StringBuilder` class and its comparison with the `String` class.

Let's work with the methods of the class `StringBuilder`. Because `StringBuilder` represents a mutable sequence of characters, the main operations on `StringBuilder` are related to the modification of its value by adding another value at the end or at a particular position, deletion of characters, or changing characters at a particular position.

4.2.1 ***The `StringBuilder` class is mutable***

In contrast to the class `String`, the class `StringBuilder` uses a non-final char array to store its value. Following is a partial definition of the class `AbstractStringBuilder` (the base class of class `StringBuilder`). It includes the declaration of the variables `value` and `count`, which are used to store the value of `StringBuilder` and its length respectively (the relevant code is in bold):

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
    /**
     * The value is used for character storage.
     */
    char value[];

    /**
     * The count is the number of characters used.
     */
    int count;
    //... rest of the code
}
```

This information will come in handy when we discuss the methods of class `StringBuilder` in the following sections.

4.2.2 ***Creating `StringBuilder` objects***

You can create objects of class `StringBuilder` using multiple overloaded constructors, as follows:

```
class CreateStringBuilderObjects {
    public static void main(String args[]) {
        StringBuilder sb1 = new StringBuilder(); ← ① No-argument constructor
        StringBuilder sb2 = new StringBuilder(sb1); ← ② Constructor that accepts a StringBuilder object
        StringBuilder sb3 = new StringBuilder(50); ← ③ Constructor that accepts an int value specifying initial capacity of StringBuilder object
        StringBuilder sb4 = new StringBuilder("Shreya Gupta"); ← ④ Constructor that accepts a String
    }
}
```

Constructor that accepts a String

- ➊ constructs a `StringBuilder` object with no characters in it and an initial capacity of 16 characters.
- ➋ constructs a `StringBuilder` object that contains the same set of characters as contained by the `StringBuilder` object passed to it.
- ➌ constructs a `StringBuilder` object with no characters and an initial capacity of 50 characters.
- ➍ constructs a `StringBuilder` object with an initial value as contained by the `String` object. Figure 4.13 illustrates `StringBuilder` object `sb4` with the value `Shreya Gupta`.

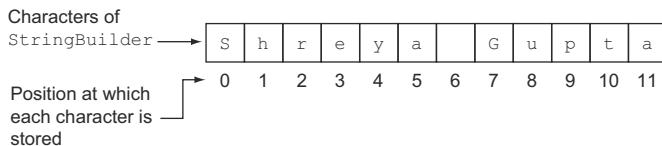


Figure 4.13 The *StringBuilder* object with character values and their corresponding storage positions

When you create a *StringBuilder* object using its default constructor, the following code executes behind the scenes to initialize the array value defined in the class *StringBuilder* itself:

```
StringBuilder() {
    value = new char[16];
}
```

Creates an array
of length 16

When you create a *StringBuilder* object by passing it a *String*, the following code executes behind the scenes to initialize the array value:

```
public StringBuilder(String str) {
    value = new char[str.length() + 16];
    append(str);
}
```

Creates an array of
length 16 + str.length

The creation of objects for the class *StringBuilder* is the basis for the next Twist in the Tale exercise. Your task in this exercise is to look up the Java API documentation or the Java source code to answer the question. You can access the Java API documentation in a couple of ways:

- View it online at <http://docs.oracle.com/javase/7/docs/api/>.
- Download it to your system from www.oracle.com/technetwork/java/javase/documentation/java-se-7-doc-download-435117.html. Accept the license agreement and click on the link for *jdk-7u6-apidocs.zip* to download it. (These links may change eventually as Oracle updates its website.)

The answer to the following Twist in the Tale exercise is given in the appendix.

Twist in the Tale 4.2

Take a look at the Java API documentation or the Java source code files and answer the following question:

Which of the following options (there's just one correct answer) correctly creates an object of the class *StringBuilder* with a default capacity of 16 characters?

- a `StringBuilder name = StringBuilder.getInstance();`
- b `StringBuilder name = StringBuilder.createInstance();`
- c `StringBuilder name = StringBuilder.buildInstance();`
- d None of the above

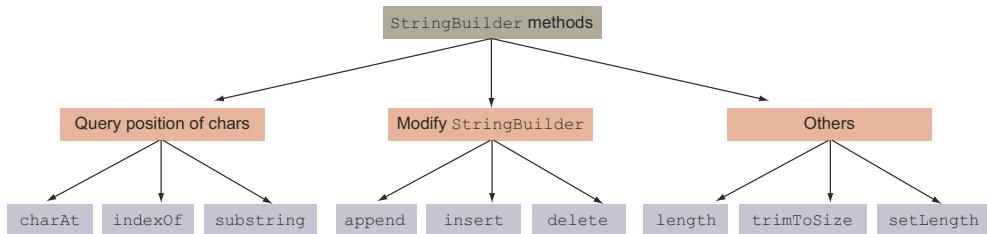


Figure 4.14 Categorization of `StringBuilder` methods

4.2.3 Methods of class `StringBuilder`

You'll be pleased to learn that a lot of the methods defined in the class `StringBuilder` work exactly like the versions in the class `String`—for example, methods such as `charAt`, `indexOf`, `substring`, and `length`. We won't discuss these again for the class `StringBuilder`. In this section, we'll discuss the other main methods of the class `StringBuilder`: `append`, `insert`, and `delete`.

Figure 4.14 shows the categorization of this class's methods.

APPEND()

The `append` method adds the specified value at the end of the existing value of a `StringBuilder` object. Because you may want to add data from multiple data types to a `StringBuilder` object, this method has been overloaded so that it can accept data of any type.

This method accepts all the primitives—`String`, `char` array, and `Object`—as method parameters, as shown in the following example:

```

class AppendStringBuilder {
    public static void main(String args[]) {
        StringBuilder sb1 = new StringBuilder();
        sb1.append(true);
        sb1.append(10);
        sb1.append('a');
        sb1.append(20.99);
        sb1.append("Hi");
        System.out.println(sb1);
    }
}
  
```

Appends boolean
Appends int
Appends char
Appends double
Appends String
Prints true10a20.99Hi

You can append a complete `char` array, `StringBuffer`, or `String` or its subset as follows:

```

StringBuilder sb1 = new StringBuilder();
char[] name = {'J', 'a', 'v', 'a', '7'};
sb1.append(name, 1, 3);
System.out.println(sb1);
  
```

Starting with position 1
append 3 characters,
position 1 inclusive
Prints ava

Because the method `append` also accepts a method parameter of type `Object`, you can pass it any object from the Java API or your own user-defined object:

```

class AppendStringBuilder2 {
    public static void main(String args[]) {
        StringBuilder sb1 = new StringBuilder();
        sb1.append("Java");
        sb1.append(new Person("Oracle"));
        System.out.println(sb1);
    }
}
class Person {
    String name;
    Person(String str) { name = str; }
}

```

The output of the previous code is:

```
JavaPerson@126b249
```

In this output, the hex value (126b249) that follows the @ sign may differ on your system.

When you append an object's value to a *StringBuilder*, the method `append` calls the target class's `toString` method to retrieve the object's String representation. If the `toString` method has been overridden by the class, then the method `append` adds the String value returned by it to the target *StringBuilder* object. In the absence of the overridden `toString` method, the `toString` method defined in the class `Object` executes. The default implementation of the method `toString` in the class `Object` returns the name of the class followed by the @ char and unsigned hexadecimal representation of the hash code of the object (the value returned by the object's `hashCode` method).



EXAM TIP For classes that haven't overridden the `toString` method, the `append` method appends the output from the default implementation of method `toString` defined in class `Object`.

It's interesting to take a quick look at how the `append` method works for the class `StringBuilder`. Following is a partial code listing of the method `append` that accepts a boolean parameter (as explained in the comments):

```

public AbstractStringBuilder append(boolean b) {
    if (b) {
        int newCount = count + 4;
        if (newCount > value.length)
            expandCapacity(newCount);
        value[count++] = 't';
        value[count++] = 'r';
        value[count++] = 'u';
        value[count++] = 'e';
    } else {
        ...
    }
    return this;
}

```

① and ② determine whether the array value can accommodate four additional characters corresponding to the boolean literal value true. It increases the capacity of the array value (used to store the characters of a `StringBuilder` object) if it isn't big enough. ③ adds individual characters of the boolean value true to the array value.

INSERT()

The `insert` method is as powerful as the `append` method. It also exists in multiple flavors (read: overloaded methods) that accept any data type. The main difference between the `append` and `insert` methods is that the `insert` method enables you to insert the requested data at a particular position, but the `append` method only allows you to add the requested data at the end of the `StringBuilder` object:

```
class InsertStringBuilder {
    public static void main(String args[]) {
        StringBuilder sb1 = new StringBuilder("Bon");
        sb1.insert(2, 'r');
        System.out.println(sb1);
    }
}
```

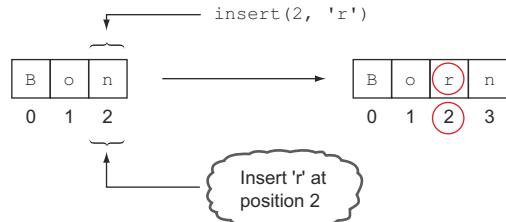


Figure 4.15 Inserting a char using the method `insert` in `StringBuilder`

Figure 4.15 illustrates the previous code.

As with `String` objects, the first character of `StringBuilder` is stored at position 0. Hence, the previous code inserts the letter r at position 2, which is occupied by the letter n. You can also insert a complete char array, `StringBuffer`, or `String` or its subset, as follows:

```
StringBuilder sb1 = new StringBuilder("123");
char[] name = {'J', 'a', 'v', 'a'};
sb1.insert(1, name, 1, 3);
```

← **Prints 1ava23**

← **Insert at sb1 position 1,
values ava from String name**

Figure 4.16 illustrates the previous code.

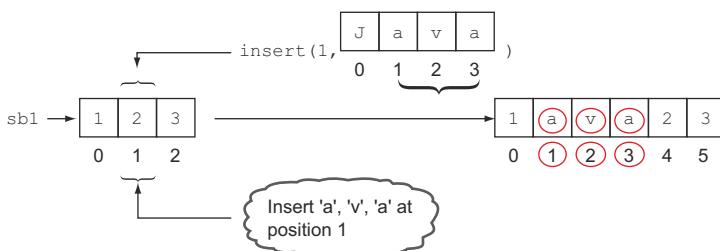


Figure 4.16 Inserting a substring of `String` in `StringBuilder`



EXAM TIP Take note of the start and end positions when inserting a value in a *StringBuilder*. Multiple flavors of the *insert* method defined in *StringBuilder* may confuse you because they can be used to insert either single or multiple characters.

DELETE() AND DELETECHARAT()

The method *delete* removes the characters in a substring of the specified *StringBuilder*. The method *deleteCharAt* removes the char at the specified position. Here's an example showing the method *delete*:

```
class DeleteStringBuilder {
    public static void main(String args[]) {
        StringBuilder sb1 = new StringBuilder("0123456");
        sb1.delete(2, 4);
        System.out.println(sb1);
    }
}
```

① Removes characters at positions starting from 2 to 4, excluding 4

Prints 01456

① removes characters at positions 2 and 3. The *delete* method doesn't remove the letter at position 4. Figure 4.17 illustrates the previous code.

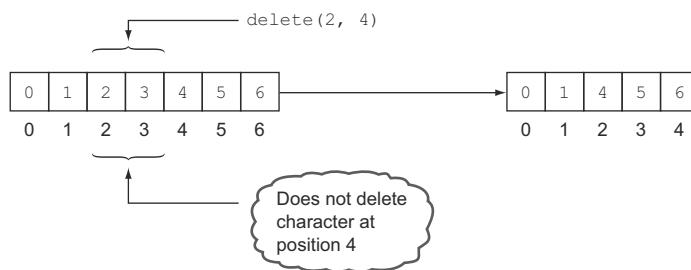


Figure 4.17 The method *delete(2, 4)* doesn't delete the character at position 4.

The method *deleteCharAt* is simple. It removes a single character, as follows:

```
class DeleteStringBuilder {
    public static void main(String args[]) {
        StringBuilder sb1 = new StringBuilder("0123456");
        sb1.deleteCharAt(2);
        System.out.println(sb1);
    }
}
```

Deletes character at position 2

Prints 013456



EXAM TIP Combinations of the *deleteCharAt* and *insert* methods can be quite confusing.

TRIM()

Unlike the class *String*, the class *StringBuilder* doesn't define the method *trim*. An attempt to use it with this class will prevent your code from compiling. The only reason I'm describing a nonexistent method here is to ward off any confusion.

REVERSE()

As the name suggests, the *reverse* method reverses the sequence of characters of a *StringBuilder*:

```
class ReverseStringBuilder {
    public static void main(String args[]) {
        StringBuilder sb1 = new StringBuilder("0123456");
        sb1.reverse();
        System.out.println(sb1);
    }
}
```

Prints
6543210



EXAM TIP You can't use the method `reverse` to reverse a substring of `StringBuilder`.

REPLACE()

Unlike the `replace` method defined in the class `String`, the `replace` method in the class `StringBuilder` replaces a sequence of characters, identified by their positions, with another `String`, as in the following example:

```
class ReplaceStringBuilder {
    public static void main(String args[]) {
        StringBuilder sb1 = new StringBuilder("0123456");
        sb1.replace(2, 4, "ABCD");
        System.out.println(sb1);
    }
}
```

Prints
0IABCD456

Figure 4.18 shows a comparison of the `replace` methods defined in the classes `String` and `StringBuilder`.

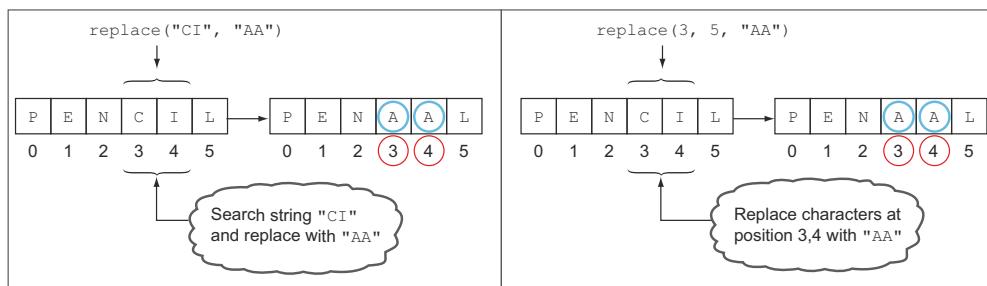


Figure 4.18 Comparing the `replace` methods in `String` (left) and `StringBuilder` (right). The method `replace` in `String` accepts the characters to be replaced. The method `replace` in `StringBuilder` accepts a position to be replaced.

SUBSEQUENCE()

Apart from using the method `substring`, you can also use the method `subSequence` to retrieve a subsequence of a `StringBuilder` object. This method returns objects of type `CharSequence`:

```
class SubSequenceStringBuilder {
    public static void main(String args[]) {
        StringBuilder sb1 = new StringBuilder("0123456");
        System.out.println(sb1.subSequence(2, 4));
        System.out.println(sb1);
    }
}
```

Prints 23
Prints 0123456

The method `subsequence` doesn't modify the existing value of a `StringBuilder` object.

4.2.4 A quick note on the class `StringBuffer`

Though the OCA Java SE 7 Programmer I exam objectives don't mention the class `StringBuffer`, you may see it in the list of (incorrect) answers in the OCA exam.

The classes `StringBuffer` and `StringBuilder` offer the same functionality, with one difference: the methods of the class `StringBuffer` are synchronized where necessary, whereas the methods of the class `StringBuilder` aren't. What does this mean? When you work with the class `StringBuffer`, only one thread out of multiple threads can execute your method. This arrangement prevents any inconsistencies in the values of the instance variables that are modified by these (synchronized) methods. But it introduces additional overhead, so working with synchronized methods and the `StringBuffer` class affects the performance of your code.

The class `StringBuilder` offers the same functionality as offered by `StringBuffer`, minus the additional feature of synchronized methods. Often your code won't be accessed by multiple threads, so it won't need the overhead of thread synchronization. If you need to access your code from multiple threads, use `StringBuffer`; otherwise use `StringBuilder`.

4.3 Arrays



[4.1] Declare, instantiate, initialize, and use a one-dimensional array



[4.2] Declare, instantiate, initialize, and use a multidimensional array

In this section, I'll cover declaration, allocation, and initialization of one-dimensional and multidimensional arrays. You'll learn about the differences between arrays of primitive data types and arrays of objects.

4.3.1 What is an array?

An array is an object that stores a collection of values. The fact that an array itself is an object is often overlooked. I'll reiterate: an array is an object itself, which implies that it stores references to the data it stores. Arrays can store two types of data:

- A collection of primitive data types
- A collection of objects

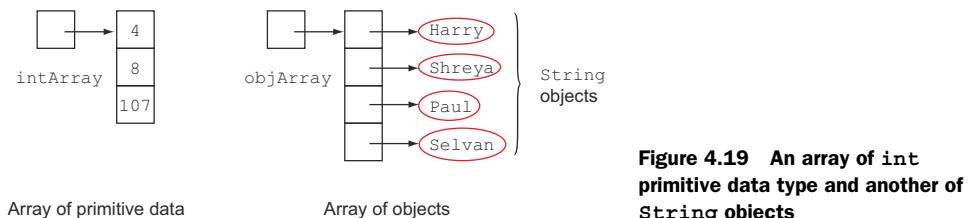
An array of primitives stores a collection of values that constitute the primitive values themselves. (With primitives, there are no objects to reference.) An array of objects stores a collection of values, which are in fact heap-memory addresses or pointers. The addresses point to (reference) the object instances that your array is said to store, which means that object arrays store references (to objects) and primitive arrays store primitive values.

The members of an array are defined in contiguous (continuous) memory locations and hence offer improved access speed. (You should be able to quickly access all the students of a class if they all can be found next to each other.)

The following code creates an array of primitive data and an array of objects:

```
class CreateArray {
    public static void main(String args[]) {
        int intArray[] = new int[] {4, 8, 107};
        String objArray[] = new String[] {"Harry", "Shreya",
                                         "Paul", "Selvan"};
    }
}
```

I'll discuss the details of creating arrays shortly. The previous example shows one of the ways to create arrays. Figure 4.19 illustrates the arrays `intArray` and `objArray`. Unlike `intArray`, `objArray` stores references to `String` objects.



NOTE Arrays are objects and refer to a collection of primitive data types or other objects.

In Java, you can define one-dimensional and multidimensional arrays. A *one-dimensional array* is an object that refers to a collection of scalar values. A two-dimensional (or more) array is referred to as a *multidimensional array*. A two-dimensional array refers to a collection of objects in which each of the objects is a one-dimensional array. Similarly, a three-dimensional array refers to a collection of two-dimensional arrays, and so on. Figure 4.20 depicts a one-dimensional array and multidimensional arrays (two-dimensional and three-dimensional).

Note that multidimensional arrays may or may not contain the same number of elements in each row or column, as shown in the two-dimensional array in figure 4.20.

Creating an array involves three steps, as follows:

- Declaring the array
- Allocating the array
- Initializing the array elements

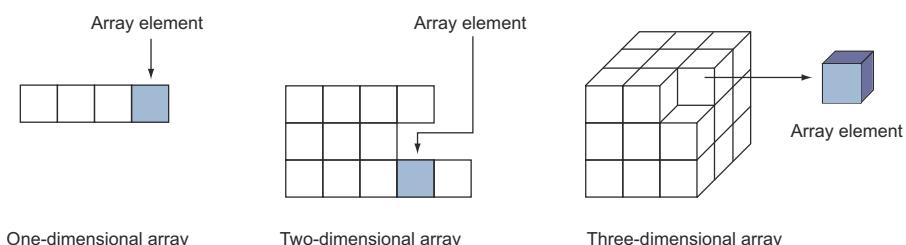


Figure 4.20 One-dimensional and multidimensional (two- and three-dimensional) arrays

You can create an array by executing the previous steps using separate lines of code or you can combine these steps on the same line of code. Let's start with the first approach: completing each step on a separate line of code.

4.3.2 Array declaration

An array declaration includes the array *type* and array *variable*, as shown in figure 4.21. The type of objects that an array can store depends on its type. An array type is followed by empty pairs of square brackets [].

To declare an array, specify its type, followed by the name of the array variable. Here's an example of declaring arrays of int and String values:

```
int intArray[];
String[] strArray;
int[] multiArray;
```

| |
| One-dimensional |
| array |
| |
| |
| Multidimensional |
| array |
| |
| |

The number of bracket pairs indicates the depth of array nesting. Java doesn't impose any theoretical limit on the level of array nesting. The square brackets can follow the array type or its name, as shown in figure 4.22.

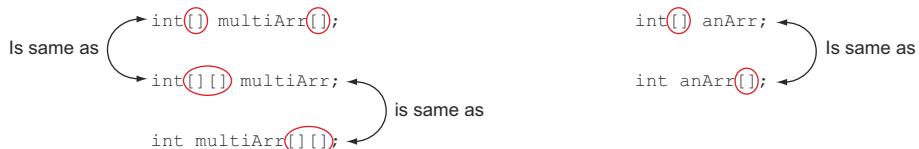


Figure 4.22 Square brackets can follow either the variable name or its type. In the case of multidimensional arrays, it can follow both of them.

The array declaration only creates a variable that refers to null, as shown in figure 4.23.

Because no elements of an array are created when it's declared, it's invalid to define the size of an array with its declaration. The following code won't compile:

```
int intArray[2];
String[5] strArray;
int[2] multiArray[3];
```

| |
| Array size can't be defined with the array
| declaration. This code won't compile.
| |

An array type can be any of the following:

- Primitive data type
- Interface
- Abstract class
- Concrete class

We declared an array of an int primitive type and a concrete class String previously. I'll discuss some complex examples with abstract classes and interfaces in section 4.3.7.

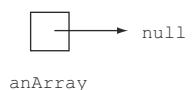


Figure 4.23 Array declaration creates a variable that refers to null.



NOTE Arrays can be of any data type other than null.

4.3.3 Array allocation

As the name suggests, array allocation will allocate memory for the elements of an array. When you allocate memory for an array, you should specify its dimensions, such as the number of elements the array should store. Note that the size of an array can't expand or reduce once it is allocated. Here are a few examples:

```
int intArray[];  
String[] strArray;  
int[] multiArr[];  
  
intArray = new int[2];  
strArray = new String[4];  
multiArr = new int[2][3];
```

Array declaration

Note use of keyword new to allocate an array

Because an array is an object, it's allocated using the keyword new, followed by the type of value that it stores, and then its size. The code won't compile if you don't specify the size of the array or if you place the array size on the left of the = sign, as follows:

```
intArray = new int[];  
intArray[2] = new int;
```

Won't compile. Array size missing.

Won't compile. Array size placed incorrectly.

The size of the array should evaluate to an int value. You can't create an array with its size specified as a floating-point number. The following line of code won't compile:

```
intArray = new int[2.4];
```

Won't compile. Can't define size of an array as a floating-point number

Java accepts an expression to specify the size of an array, as long as it evaluates to an int value. The following are valid array allocations:

```
strArray = new String[2*5];  
int x = 10, y = 4;  
strArray = new String[x*y];  
strArray = new String[Math.max(2, 3)];
```

2*5 evaluates to an integer value.

This is acceptable. Expression x*y evaluates to an integer value.

This is acceptable. Math.max(2,3) returns an int value.

Let's allocate the multidimensional array multiArr, as follows:

```
int[] multiArr[];  
multiArr = new int[2][3];
```

Array declaration

OK to define size in both the square brackets

You can also allocate the multidimensional array multiArr by defining size in only the first square bracket:

```
multiArr = new int[2][];
```

OK to define the size in only the first square brackets

It's interesting to note what happens when the multidimensional array multiArr is allocated by defining sizes for a single dimension and for both its dimensions. This difference is shown in figure 4.24.

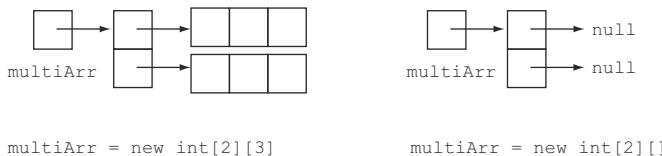


Figure 4.24 The difference in array allocation of a two-dimensional array when it's allocated using values for only one of its dimensions and for both its dimensions.

You can't allocate a multidimensional array as follows:

```

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

②

Size in first square bracket missing

①

Multidimensional array declaration
Nonmatching square brackets

- ① won't compile because there's a mismatch in the number of square brackets on both sides of the assignment operator (=). The compiler required [] [] on the right side of the assignment operator, but it finds only []. ② won't compile because you can't allocate a multidimensional array without including a size in the first square brackets and defining a size in the second square brackets.

Once allocated, the array elements store their default values. For arrays that store objects, all the allocated array's elements store null. For arrays that store primitive values, the default values depend on the exact data types stored by them.



EXAM TIP Once allocated, all the array elements store their default values. Elements in an array that store objects default to null. Elements of an array that store primitive data types store 0 for integer types (byte, short, int, long), 0.0 for decimal types (float and double), false for boolean, or \u0000 for char data.

4.3.4 Array initialization

You can initialize an array as follows:

```

int intArray[];
intArray = new int[2];

for (int i=0; i<intArray.length; i++) {
    intArray[i] = i + 5;
}

intArray[0] = 10;
intArray[1] = 1870;
  
```

Array declaration

Array allocation

① Initializes array using a for loop

② Reinitializes individual array elements

- ① Uses a for loop to initialize the array intArray with the required values. ② initializes the individual array elements without using a for loop. Note that all array objects can access the instance variable length, which stores the array size.

Similarly, a String array can be declared, allocated, and initialized as follows:

```

String[] strArray;
strArray = new String[4];

for (int i=0; i<strArray.length; i++) {
    strArray[i] = new String("Hello" + i);
}
  
```

Array declaration

Array allocation

Initializes array using a for loop

```
strArray[1] = "Summer";
strArray[3] = "Winter";
strArray[0] = "Autumn";
strArray[2] = "Spring";
```

Initializes array without using a for loop

When you initialize a two-dimensional array, you can use nested for loops to initialize its array elements. Also notice that to access an element in a two-dimensional array, you should use two array position values, as follows:

```
int[] multiArr[];
multiArr = new int[2][3];

for (int i=0; i<multiArr.length; i++) {
    for (int j=0; j<multiArr[i].length; j++) {
        multiArr[i][j] = i + j;
    }
}

multiArr[0][0] = 10;
multiArr[1][2] = 1210;
multiArr[0][1] = 110;
multiArr[0][2] = 1087;
```

Array declaration
←
Array allocation

Initializes array using a for loop

Initializes array without using a for loop

What happens when you try to access a nonexistent array index position? The following code creates an array of size 2 but tries to access its array element at index 3:

```
int intArray[] = new int[2];
System.out.println(intArray[3]);
```

Length of intArray is 2
←
3 isn't a valid index position for intArray

The previous code will throw a runtime exception, `ArrayIndexOutOfBoundsException`. For an array of size 2, the only valid index positions are 0 and 1. All the rest of the array index positions will throw the exception `ArrayIndexOutOfBoundsException` at runtime.



NOTE Don't worry if you can't immediately absorb all of the information related to exceptions here. Exceptions are covered in detail in chapter 7.

The Java compiler doesn't check the range of the index positions at which you try to access an array element. You may be surprised to learn that the following line of code will compile successfully even though it uses a negative array index value:

```
int intArray[] = new int[2];
System.out.println(intArray[-10]);
```

Length of intArray is 2
←
Will compile successfully even though it tries to access array element at negative index

Though the previous code compiles successfully, it will throw the exception `ArrayIndexOutOfBoundsException` at runtime. Code to access an array element will fail to compile if you don't pass it a `char`, `byte`, `short`, or `int` data type (wrapper classes are not on this exam, and I don't include them in this discussion):

```
int intArray[] = new int[2];
System.out.println(intArray[1.2]);
```

Won't compile—can't specify array index using floating-point number



EXAM TIP Code to access an array index will throw a runtime exception if you pass it an invalid array index value. Code to access an array index will fail to compile if you don't use a char, byte, short, or int.

Also, you can't remove array positions. For an array of objects, you can set a position to value null, but it doesn't remove the array position:

```
String[] strArray = new String[] {"Autumn", "Summer",
                                 "Spring", "Winter"};
strArray[2] = null;
for (String val : strArray)
    System.out.println(val);
```

1 Define an array of String objects

2 Can you remove an array position like this?

3 Outputs four values

① creates an array of String and initializes it with four String values. ② sets the value at array index 2 to null. ③ iterates over all the array elements. As shown in the following output, four (not three) values are printed:

```
Autumn
Summer
null
Winter
```

4.3.5 Combining array declaration, allocation, and initialization

You can combine all the previously mentioned steps of array declaration, allocation, and initialization into one step, as follows:

```
int intArray[] = {0, 1};
String[] strArray = {"Summer", "Winter"};
int multiArray[][] = {{0, 1}, {3, 4, 5}};
```

Notice that the previous code

- Doesn't use the keyword new to initialize an array
- Doesn't specify the size of the array
- Uses a single pair of braces to define values for a one-dimensional array and multiple pairs of braces to define a multidimensional array

All the previous steps of array declaration, allocation, and initialization can be combined in the following way, as well:

```
int intArray2[] = new int[]{0, 1};
String[] strArray2 = new String[]{"Summer", "Winter"};
int multiArray2[][] = new int[][]{{0, 1}, {3, 4, 5}};
```

Unlike the first approach, the preceding code uses the keyword new to initialize an array.

If you try to specify the size of an array with the preceding approach, the code won't compile. Here are a few examples:

```
int intArray2[] = new int[2]{0, 1};
String[] strArray2 = new String[2>{"Summer", "Winter"};
int multiArray2[][] = new int[2][]{{0, 1}, {3, 4, 5}};
```



EXAM TIP When you combine an array declaration, allocation, and initialization in a single step, you can't specify the size of the array. The size of the array is calculated by the number of values that are assigned to the array.

Another important point to note is that if you declare and initialize an array using two separate lines of code, you'll use the keyword new to initialize the values. The following lines of code are correct:

```
int intArray[];
intArray = new int[] {0, 1};
```

But you can't miss the keyword new and initialize your array as follows:

```
int intArray[];
intArray = {0, 1};
```

4.3.6 Asymmetrical multidimensional arrays

At the beginning of this section, I mentioned that a multidimensional array can be asymmetrical. Arrays can define a different number of columns for each of its rows.

The following example is an asymmetrical two-dimensional array:

```
String multiStrArr[][] = new String[][] {
    {"A", "B"},
    null,
    {"Jan", "Feb", "Mar"},
};
```

Figure 4.25 shows this asymmetrical array.

As you might have noticed, `multiStrArr[1]` refers to a null value. An attempt to access any element of this array, such as `multiStrArr[1][0]`, will throw an exception. This brings us to the next Twist in the Tale exercise (answers are in the appendix).

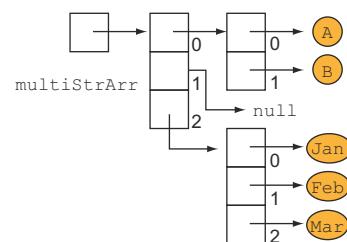


Figure 4.25 An asymmetrical array

Twist in the Tale 4.3

Modify some of the code used in the previous example as follows:

```
Line1> String multiStrArr[][] = new String[][] {
Line2>                               {"A", "B"},
Line3>                               null,
Line4>                               {"Jan", "Feb", null},
Line5>                               };
```

Which of the following individual options are true for the previous code?

- a Code on line 4 is the same as `{"Jan", "Feb", null, null}`,
- b No value is stored at `multiStrArr[2][2]`
- c No value is stored at `multiStrArr[1][1]`
- d Array `multiStrArr` is asymmetric.

4.3.7 Arrays of type interface, abstract class, and class Object

In the section on array declaration, I mentioned that the type of an array can also be an interface or an abstract class. What values do elements of these arrays store? Let's take a look at some examples.

INTERFACE TYPE

If the type of an array is an interface, its elements are either null or objects that implement the relevant interface type. For example, for the interface `MyInterface`, the array `interfaceArray` can store references to objects of either the class `MyClass1` or `MyClass2`:

```
interface MyInterface {}
class MyClass1 implements MyInterface {}
class MyClass2 implements MyInterface {}

class Test {
    MyInterface[] interfaceArray = new MyInterface[]
    {
        new MyClass1(),
        null,
        new MyClass2()
    };
}
```

ABSTRACT CLASS TYPE

If the type of an array is an abstract class, its elements are either null or objects of concrete classes that extend the relevant abstract class:

```
abstract class Vehicle{}
class Car extends Vehicle {}
class Bus extends Vehicle {}

class Test {
    Vehicle[] vehicleArray = { new Car(),
                               new Bus(),
                               null};
```



null is a valid element

Next, I'll discuss a special case in which the type of an array is `Object`.

OBJECT

Because all classes extend the class `java.lang.Object`, elements of an array whose type is `java.lang.Object` can refer to any object. Here's an example:

```
interface MyInterface {}
class MyClass1 implements MyInterface {}
abstract class Vehicle{}
class Car extends Vehicle {}
class Test {
    Object[] objArray = new Object[] {
        new MyClass1(),
        null,
        new Car(),
        new java.util.Date(),
```



null is a valid element

```

        new String("name"),
        new Integer [7]
    };
}

```

① Array element of type Object
can refer to another array

① is valid code. Because an array is an object, the element of the array of `java.lang.Object` can refer to another array. Figure 4.26 illustrates the previously created array, `objArray`.

4.3.8 Members of an array

Array objects have the following public members:

- `length`—The variable `length` contains the number of components of the array.
- `clone()`—This method overrides the method `clone` defined in class `Object` but doesn't throw checked exceptions. The return type of this method is the same as the array's type. For example, for an array of type `Type[]`, this method returns `Type[]`.
- *Inherited methods*—Methods inherited from the class `Object`, except the method `clone`.

As mentioned in the earlier section on the `String` class, a `String` uses method `length()` to retrieve its length. With an array, you can use the array's variable `length` to determine the number of the array's elements. In the exam, you may be tricked by code that tries to access the `length` of a `String` using variable `length`. Note the correct combination of class and member used to access its length:

- `string`—Retrieve length using the method `length()`
- `Array`—Determine element count using the variable `length`

I have an interesting way to remember this rule. As opposed to an array, you'll invoke a *lot* of methods on `String` objects. So you use *method* `length()` to retrieve the length of `String` and *variable* `length` to retrieve the length of an array.

4.4 ArrayList



[4.3] Declare and use an `ArrayList`

In this section, I'll cover how to use `ArrayList`, its commonly used methods, and the advantages it offers over an array.

The OCA Java SE 7 Programmer I exam covers only one class from the Java Collection API: `ArrayList`. The rest of the classes from the Java Collection API are covered in the OCP Java SE 7 Programmer II exam (exam number 1Z0-804). One of the reasons

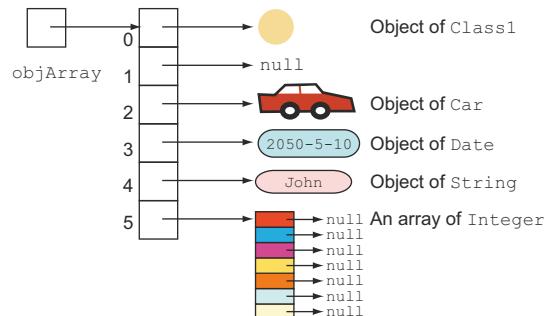


Figure 4.26 An array of class Object

to include this class in the Java Associate exam could be how frequently this class is used by all Java programmers.

ArrayList is one of the most widely used classes from the Collections framework. It offers the best combination of features offered by an *array* and the *List* data structure. The most commonly used operations with a list are: add items to a list, modify items in a list, delete items from a list, and iterate over the items.

One frequently asked question by Java developers is: “Why should I bother with an ArrayList when I can already store objects of the same type in an array?” The answer lies in the ease of use of an ArrayList. This is an important question, and the OCA Java SE 7 Programmer I exam contains explicit questions on the practical reasons for using an ArrayList.

You can compare an ArrayList with a resizable array. As you know, once it’s created, you can’t increase or decrease the size of an array. On the other hand, an ArrayList automatically increases and decreases in size as elements are added to or removed from it. Also, unlike arrays, you don’t need to specify an initial size to create an ArrayList.

Let’s compare an ArrayList and an array with real-world objects. Just as a balloon can increase and decrease in size when it’s inflated or deflated, an ArrayList can increase or decrease in size as values are added to it or removed from it. One comparison is a cricket ball, as it has a predefined size. Once created, like an array, it can’t increase or decrease in size.

Here are a few more important properties of an ArrayList:

- It implements the interface List.
- It allows null values to be added to it.
- It implements all list operations (add, modify, and delete values).
- It allows duplicate values to be added to it.
- It maintains its insertion order.
- You can use either Iterator or ListIterator (an implementation of the Iterator interface) to iterate over the items of an ArrayList.
- It supports generics, making it type safe. (You have to declare the type of the elements that should be added to an ArrayList with its declaration.)

4.4.1 Creating an ArrayList

The following example shows you how to create an ArrayList:

```
import java.util.ArrayList;
public class CreateArrayList {
    public static void main(String args[]) {
        ArrayList<String> myArrList = new ArrayList<String>();
    }
}
```

1 Import
java.util.ArrayList

2 Declare an
ArrayList object

Package `java.util` isn't implicitly imported into your class, which means that ① imports the class `ArrayList` in the class `CreateArrayList` defined previously. To create an `ArrayList`, you need to inform Java about the type of the objects that you want to store in this collection of objects. ② declares an `ArrayList` called `myArrList`, which can store `String` objects specified by the name of the class `String` between the angle brackets (`<>`). Note that the name `String` appears twice in the code at ②, once on the left side of the equal sign and the other on the right. Do you think the second one seems redundant? Congratulations, Oracle agrees with you. Starting with Java version 7, you can omit the object type on the right side of the equal sign and create an `ArrayList` as follows:

```
ArrayList<String> myArrList = new ArrayList<>();
```

**Missing object type on
right of = works in Java
version 7 and above**

Many developers still work with Java SE versions prior to version 7, so you're likely to see some developers still using the older way of creating an `ArrayList`.

Take a look at what happens behind the scenes (in the Java source code) when you execute the previous statement to create an `ArrayList`. Because you didn't pass any arguments to the constructor of class `ArrayList`, its no-argument constructor will execute. Examine the definition of the following no-argument constructor defined in class `ArrayList.java`:

```
/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() {
    this(10);
}
```

Because you can use an `ArrayList` to store any type of `Object`, `ArrayList` defines an instance variable `elementData` of type `Object` to store all its individual elements. Following is a partial code listing from class `ArrayList`:

```
/**
 * The array buffer into which the elements of the ArrayList are stored.
 * The capacity of the ArrayList is the length of this array buffer.
 */
private transient Object[] elementData;
```

Figure 4.27 illustrates the variable `elementData` shown within an object of `ArrayList`.



Figure 4.27 Variable `elementData` shown within an object of `ArrayList`

Here is the definition of the constructor from the class ArrayList (ArrayList.java), which initializes the previously defined instance variable, elementData:

```
/**
 * Constructs an empty list with the specified initial capacity.
 *
 * @param initialCapacity the initial capacity of the list
 * @exception IllegalArgumentException if the specified initial capacity
 *          is negative
 */
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: " +
                                           initialCapacity);
    this.elementData = new Object[initialCapacity];
}
```

Wait a minute. Did you notice that an ArrayList uses an array to store its individual elements? Does that make you wonder why on earth you would need another class if it uses a type (array, to be precise) that you already know how to work with? The simple answer is that you wouldn't want to reinvent the wheel.

For an example, answer this question: to decode an image, which of the following options would you prefer:

- Creating your own class using characters to decode the image
- Using an existing class that offers the same functionality

Obviously, it makes sense to go with the second option. If you use an existing class that offers the same functionality, you get more benefits with less work. The same logic applies to ArrayList. It offers you all of the benefits of using an array, with none of the disadvantages. It offers an expandable array that is modifiable.



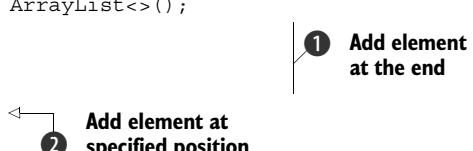
NOTE An ArrayList uses an array to store its elements. It provides you with the functionality of a dynamic array.

I'll cover how to add, modify, delete, and access the elements of an ArrayList in the following sections. Let's start with adding elements to an ArrayList.

4.4.2 Adding elements to an ArrayList

Let's start with adding elements to an ArrayList, as follows:

```
import java.util.ArrayList;
public class AddToArrayList {
    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<>();
        list.add("one");
        list.add("two");
        list.add("four");
        list.add(2, "three");
    }
}
```



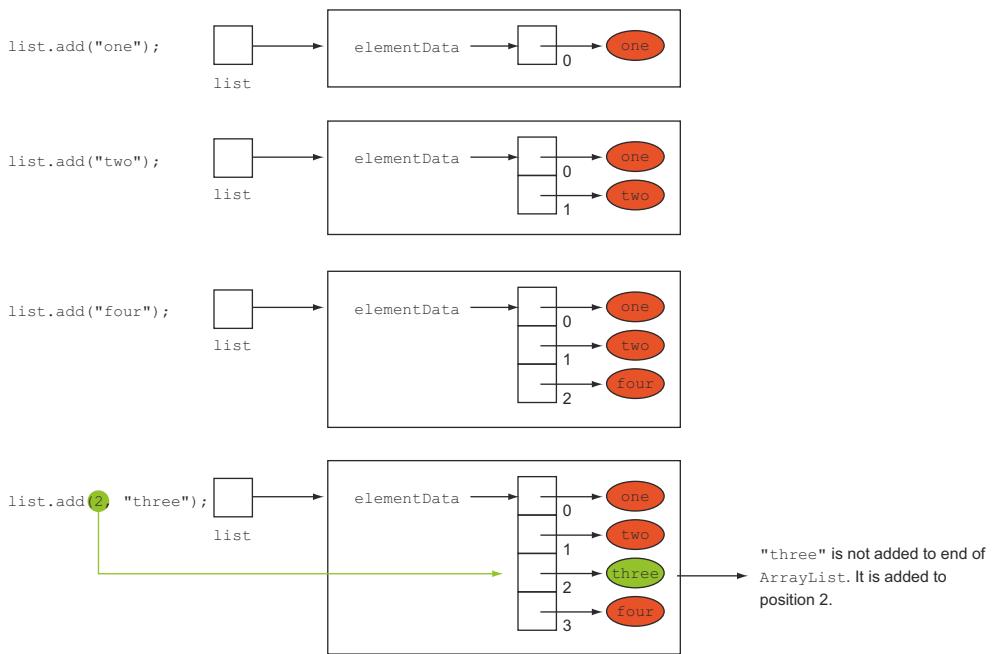


Figure 4.28 Code that adds elements to the end of an ArrayList and at a specified position

You can add a value to an ArrayList either at its end or at a specified position. ① adds elements at the end of list. ② adds an element to list at position 2. Please note that the first element of an ArrayList is stored at position 0. Hence, at ② the String literal value "three" will be inserted at position 2, which was occupied by the literal value "four". When a value is added to a place that is already occupied by another element, the values shift by a place to accommodate the newly added value. In this case, the literal value "four" shifted to position 3 to make way for the literal value "three" (see figure 4.28).

Let's see what happens behind the scenes in an ArrayList when you add an element to it. Here's the definition of the method add from the class ArrayList:

```
/**
 * Appends the specified element to the end of this list.
 *
 * @param e element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    ensureCapacity(size + 1);      // Create another array with
                                   // the increased capacity
                                   // and copy existing elements to it.
    elementData[size++] = e;       // Store the newly added variable
                                   // reference at the
                                   // end of the list.
```

```

        return true;
    }
}

```

When you add an element to the end of the list, the `ArrayList` first checks whether its instance variable `elementData` has an empty slot at the end. If there's an empty slot at its end, it stores the element at the first available empty slot. If no empty slots exist, the method `ensureCapacity` creates another array with a higher capacity and copies the existing values to this newly created array. It then copies the newly added value at the first available empty slot in the array.

When you add an element at a particular position, an `ArrayList` creates a new array and inserts all its elements at positions other than the position you specified. If there are any subsequent elements to the right of the position that you specified, it shifts them by one position. Then it adds the new element at the requested position.

PRACTICAL TIP Understanding how and why a class works in a particular manner will take you a long way, in regard to both the certification exam and your career. This understanding should help you retain the information for a longer time and help you answer questions in the certification exam that are looking to verify your practical knowledge of using this class. Last, but not least, the internal workings of a class will enable you to make informed decisions on using a particular class at your workplace and writing efficient code.

4.4.3 Accessing elements of an `ArrayList`

Before we modify or delete the elements of an `ArrayList`, let's see how to access them. To access the elements of an `ArrayList`, you can either use Java's enhanced for loop, `Iterator`, or `ListIterator`.

The following code accesses and prints all the elements of an `ArrayList` using the enhanced for loop (code to access elements is in bold):

```

import java.util.ArrayList;

public class AccessArrayList {
    public static void main(String args[]) {
        ArrayList<String> myArrList = new ArrayList<>();
        myArrList.add("One");
        myArrList.add("Two");
        myArrList.add("Four");
        myArrList.add(2, "Three");

        for (String element : myArrList) {
            System.out.println(element);
        }
    }
}

```

1 Code to access
ArrayList elements

The output of the previous code is as follows:

```

One
Two
Three
Four

```

➊ defines the enhanced for loop to access all the elements of the myArrayList.

Let's take a look at how to use a ListIterator to loop through all the values of an ArrayList:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class AccessArrayListUsingListIterator {
    public static void main(String args[]) {
        ArrayList<String> myArrayList = new ArrayList<String>();
        myArrayList.add("One");
        myArrayList.add("Two");
        myArrayList.add("Four");
        myArrayList.add(2, "Three");
        ListIterator<String> iterator = myArrayList.listIterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

➊ gets the iterator associated with ArrayList myArrayList. ➋ calls the method hasNext on iterator to check whether more elements of myArrayList exist. The method hasNext returns a boolean true value if more of its elements exist, and false otherwise. ➌ calls the method next on iterator to get the next item from myArrayList.

The previous code prints out the same results as the code that preceded it, the code that used an enhanced for loop to access ArrayList's elements. A ListIterator doesn't contain any reference to the *current* element of an ArrayList. ListIterator provides you with a method (hasNext) to check whether more elements exist for an ArrayList. If true, you can extract its *next* element using method next().

Note that an ArrayList preserves the insertion order of its elements. ListIterator and the enhanced for loop will return to you the elements in the order in which you added them.



EXAM TIP An ArrayList preserves the order of insertion of its elements. Iterator, ListIterator, and the enhanced for loop will return the elements in the order in which they were added to the ArrayList. An iterator (Iterator or ListIterator) lets you remove elements as you iterate an ArrayList. It's not possible to remove elements from an ArrayList while iterating it using a for loop.

4.4.4 **Modifying the elements of an ArrayList**

You can modify an ArrayList by either replacing an existing element in ArrayList or modifying all of its existing values. The following code uses the set method to replace an element in an ArrayList:

```
import java.util.ArrayList;

public class ReplaceElementInArrayList {
    public static void main(String args[]) {
        ArrayList<String> myArrayList = new ArrayList<String>();
```

```

myArrList.add("One");
myArrList.add("Two");
myArrList.add("Three");
myArrList.set(1, "One and Half");
for (String element:myArrList)
    System.out.println(element);
}
}

```

Replace ArrayList element at position 1 ("Two") with "One and Half"

The output of the previous code is as follows:

```

One
One and Half
Three

```

You can also modify the existing values of an ArrayList by accessing its individual elements. Because Strings are immutable, let's try this with StringBuilder. Here's the code:

```

import java.util.ArrayList;
public class ModifyArrayListWithStringBuilder {
    public static void main(String args[]) {
        ArrayList<StringBuilder> myArrList =
            new ArrayList<StringBuilder>();
        myArrList.add(new StringBuilder("One"));
        myArrList.add(new StringBuilder("Two"));
        myArrList.add(new StringBuilder("Three"));

        for (StringBuilder element : myArrList)
            element.append(element.length());

        for (StringBuilder element : myArrList)
            System.out.println(element);
    }
}

```

1 Access ArrayList elements and modify them

The output of this code is as follows:

```

One3
Two3
Three5

```

1 accesses all the elements of myArrList and modifies the element value by appending its length to it. The modified value is printed by accessing myArrList elements again.

4.4.5 Deleting the elements of an ArrayList

ArrayList defines two methods to remove its elements, as follows:

- `remove(int index)`—This method removes the element at the specified position in this list.
- `remove(Object o)`—This method removes the first occurrence of the specified element from this list, if it's present.

Let's take a look at some code that uses these removal methods:

```

import java.util.ArrayList;
public class DeleteElementsFromArrayList {
    public static void main(String args[]) {
        ArrayList<StringBuilder> myArrList = new ArrayList<>();

        StringBuilder sb1 = new StringBuilder("One");
        StringBuilder sb2 = new StringBuilder("Two");
        StringBuilder sb3 = new StringBuilder("Three");
        StringBuilder sb4 = new StringBuilder("Four");

        myArrList.add(sb1);
        myArrList.add(sb2);
        myArrList.add(sb3);
        myArrList.add(sb4);

        myArrList.remove(1);
        for (StringBuilder element : myArrList)
            System.out.println(element);

        myArrList.remove(sb3);
        myArrList.remove(new StringBuilder("Four"));

        System.out.println();
        for (StringBuilder element : myArrList)
            System.out.println(element);
    }
}

```

Doesn't remove Four

1

Remove element at position 1

Prints One, Three, and Four

Removes Three from list

Prints One and Four

The output of the previous code is as follows:

```

One
Three
Four

One
Four

```

① tries to remove the `StringBuilder` with the value "Four" from `myArrList`. The removal of the specified element fails because of the manner in which the object references are compared for equality. Two objects are equal if their object references (the variables that store them) point to the same object.

You can always override the `equals` method in your own class to change this default behavior. The following is an example using the class `MyPerson`:

```

import java.util.ArrayList;
class MyPerson {
    String name;
    MyPerson(String name) { this.name = name; }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof MyPerson) {
            MyPerson p = (MyPerson) obj;
            boolean isEqual = p.name.equals(this.name);
            return isEqual;
        }
        else
    }
}

```

Cast obj to MyPerson

1

Method equals

null and objects of type other than MyPerson can't be equal to this object

Compare name of method parameter to that of this object's name

```

        return false;
    }
}
public class DeleteElementsFromArrayList2 {
    public static void main(String args[]) {
        ArrayList<MyPerson> myArrList = new ArrayList<MyPerson>();

        MyPerson p1 = new MyPerson("Shreya");
        MyPerson p2 = new MyPerson("Paul");
        MyPerson p3 = new MyPerson("Harry");

        myArrList.add(p1);
        myArrList.add(p2);
        myArrList.add(p3);

        myArrList.remove(new MyPerson("Paul"));

        for (MyPerson element:myArrList)
            System.out.println(element.name);
    }
}

```

At ①, the method `equals` in class `MyPerson` overrides method `equals` in class `Object`. It returns `false` if a `null` value is passed to this method. It returns `true` if an object of `MyPerson` is passed to it with a matching value for its instance variable `name`.

At ②, the method `remove` removes the element with the name `Paul` from `myArrList`. As mentioned earlier, the method `remove` compares the objects for equality before removing it from `ArrayList` by calling method `equals`.

When elements of an `ArrayList` are removed, the remaining elements are rearranged at their correct positions. This change is required to retrieve all the remaining elements at their correct new positions.

4.4.6 Other methods of `ArrayList`

Let's briefly discuss the other important methods defined in `ArrayList`.

ADDING MULTIPLE ELEMENTS TO AN `ARRAYLIST`

You can add multiple elements to an `ArrayList` from another `ArrayList` or any other class that is a subclass of `Collection` by using the following overloaded versions of method `addAll`:

- `addAll(Collection<? extends E> c)`
- `addAll(int index, Collection<? extends E> c)`

Method `addAll(Collection<? extends E> c)` appends all of the elements in the specified collection to the end of this list in the order in which they're returned by the specified collection's `Iterator`. If you aren't familiar with generics, and the parameters of this method look scary to you, don't worry—other classes from the `Collection API` are not on this exam.

Method `addAll(int index, Collection<? extends E> c)` inserts all of the elements in the specified collection into this list, starting at the specified position.

In the following code example, all elements of `ArrayList` `yourArrList` are inserted into `ArrayList` `myArrList`, starting at position 1:

```
ArrayList<String> myArrList = new ArrayList<String>();
myArrList.add("One");
myArrList.add("Two");

ArrayList<String> yourArrList = new ArrayList<String>();
yourArrList.add("Three");
yourArrList.add("Four");
myArrList.addAll(1, yourArrList);
for (String val : myArrList)
    System.out.println(val);
```

Add elements of yourArrList to myArrList

The output of the previous code is as follows:

```
One
Three
Four
Two
```

The elements of `yourArrList` aren't removed from it. The objects that are stored in `yourArrList` can now be referred to from `myArrList`.

What happens if you modify the *common* object references in these lists, `myArrList` and `yourArrList`? We have two cases here: In the first one, you *reassign* the object reference using either of the lists. In this case, the value in the second list will remain unchanged. In the second case, you *modify* the internals of any of the common list elements—in this case, the change will be reflected in both of the lists.



EXAM TIP This is also one of the favorite topics of the exam authors. In the exam, you're likely to encounter a question that adds the same object reference to multiple lists and then tests you on your understanding of the state of the same object and reference variable in all the lists. If you have any questions on this issue, please refer to the section on reference variables (section 2.3).

Time for our next Twist in the Tale exercise. Let's modify some of the code that we've used in our previous examples and see how it affects the output (answers in the appendix).

Twist in the Tale 4.4

What is the output of the following code?

```
ArrayList<String> myArrList = new ArrayList<String>();
String one = "One";
String two = new String("Two");
myArrList.add(one);
myArrList.add(two);
ArrayList<String> yourArrList = myArrList;
one.replace("O", "B");
```

```

for (String val : myArrList)
    System.out.print(val + ":");

for (String val : yourArrList)
    System.out.print(val + ":");

a One:Two:One:Two:
b Bne:Two:Bne:Two:
c One:Two:Bne:Two:
d Bne:Two:One:Two:

```

CLEARING ARRAYLIST ELEMENTS

You can remove all the ArrayList elements by calling clear on it. Here's an example:

```

ArrayList<String> myArrList = new ArrayList<String>();
myArrList.add("One");
myArrList.add("Two");

myArrList.clear();

for (String val:myArrList)
    System.out.println(val);

```

The previous code won't print out anything because there are no more elements in myArrList.

ACCESSING INDIVIDUAL ARRAYLIST ELEMENTS

In this section, we'll cover the following methods for accessing elements of an ArrayList:

- `get(int index)`—This method returns the element at the specified position in this list.
- `size()`—This method returns the number of elements in this list.
- `contains(Object o)`—This method returns true if this list contains the specified element.
- `indexOf(Object o)`—This method returns the index of the first occurrence of the specified element in this list, or -1 if this list doesn't contain the element.
- `lastIndexOf(Object o)`—This method returns the index of the last occurrence of the specified element in this list, or -1 if this list doesn't contain the element.

You can retrieve an object at a particular position in ArrayList and determine its size as follows:

```

ArrayList<String> myArrList = new ArrayList<String>();
myArrList.add("One");
myArrList.add("Two");
String valFromList = myArrList.get(1);
System.out.println(valFromList);           ↪ Prints Two—element
                                            at position 1
System.out.println(myArrList.size());       ↪ Prints 2

```

Behind the scenes, the method `get` will check whether the requested position exists in the `ArrayList` by comparing it with the array's size. If the requested element isn't within the range, the `get` method throws a `java.lang.IndexOutOfBoundsException` error at runtime.

All of the remaining three methods—`contains`, `indexOf`, and `lastIndexOf`—require you to have an unambiguous and strong understanding of how to determine the equality of objects. `ArrayList` stores objects, and these three methods will compare the values that you pass to these methods with all the elements of the `ArrayList`.

By default, objects are considered equal if they are referred to by the same variable (the `String` class is an exception with its pool of `String` objects). If you want to compare objects by their state (values of the instance variable), override the `equals` method in that class. I've already demonstrated the difference in how equality of objects of a class is determined, when the class overrides its `equals` method and when it doesn't, in section 4.4.5 with an overridden `equals` method in the class `MyPerson`.

Let's see the usage of all these methods:

```
public class MiscMethodsArrayList3 {
    public static void main(String args[]) {
        ArrayList<StringBuilder> myArrList =
            new ArrayList<StringBuilder>();
        StringBuilder sb1 = new StringBuilder("Jan");
        StringBuilder sb2 = new StringBuilder("Feb");
        myArrList.add(sb1);
        myArrList.add(sb2);
        myArrList.add(sb2);
        System.out.println(myArrList.contains(new StringBuilder("Jan")));
        System.out.println(myArrList.contains(sb1));
        System.out.println(myArrList.indexOf(new StringBuilder("Feb")));
        System.out.println(myArrList.indexOf(sb2));
        System.out.println(myArrList.lastIndexOf(
            new StringBuilder("Feb")));
        System.out.println(myArrList.lastIndexOf(sb2));
    }
}
```

The output of the previous code is as follows:

```
false
true
-1
1
-1
2
```

Take a look at the output of the same code using a list of `MyPerson` objects that has overridden the `equals` method. First, here's the definition of the class `MyPerson`:

```

class MyPerson {
    String name;
    MyPerson(String name) { this.name = name; }
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof MyPerson) {
            MyPerson p = (MyPerson) obj;
            boolean isEqual = p.name.equals(this.name);
            return isEqual;
        }
        else
            return false;
    }
}

```

Overridden equals method in class MyPerson. It returns true for same String values for instance variable name.

The definition of the class MiscMethodsArrayList4 follows:

```

public class MiscMethodsArrayList4 {
    public static void main(String args[]) {
        ArrayList<MyPerson> myArrList = new ArrayList<MyPerson>();

        MyPerson p1 = new MyPerson("Shreya");
        MyPerson p2 = new MyPerson("Paul");
        myArrList.add(p1);           ← Adds p1 to ArrayList
        myArrList.add(p2);           ← Adds p2 to ArrayList
        myArrList.add(p2);           ← Adds p2 to ArrayList again
    }

    Prints true
    System.out.println(myArrList.contains(new MyPerson("Shreya")));
    System.out.println(myArrList.contains(p1));
    Prints true

    Prints 1
    System.out.println(myArrList.indexOf(new MyPerson("Paul")));
    System.out.println(myArrList.indexOf(p2));
    Prints 1

    Prints 2
    System.out.println(myArrList.lastIndexOf(new MyPerson("Paul")));
    System.out.println(myArrList.lastIndexOf(p2));
    Prints 2
}

```

As you can see from the output of the preceding code, equality of the objects of class MyPerson is determined by the rules defined in its equals method. Two objects of class MyPerson with the same value for its instance variable name are considered to be equal. myArrList stores objects of class MyPerson. To find a target object, myArrList will rely on the output given by the equals method of class MyPerson; it won't compare the object references of the stored and target objects.



EXAM TIP An ArrayList can accept duplicate object values.

CLONING AN ARRAYLIST

The method `clone` defined in the class `ArrayList` returns a *shallow copy* of this `ArrayList` instance. “Shallow copy” means that this method creates a new instance of the `ArrayList` object to be cloned. Its element references are copied, but the objects themselves are not.

Here's an example:

```

public class MiscMethodsArrayList5 {
    public static void main(String args[]) {
        ArrayList<StringBuilder> myArrList = new ArrayList<StringBuilder>();
        StringBuilder sb1 = new StringBuilder("Jan");
        StringBuilder sb2 = new StringBuilder("Feb");

        myArrList.add(sb1);
        myArrList.add(sb2);
        myArrList.add(sb2);

        ArrayList<StringBuilder> assignedArrList = myArrList; 1 Assigns object referred to by myArrList to assignedArrList

        Prints true 3 ↗ System.out.println(myArrList == assignedArrList); 2 Clones myArrList and assigns it to clonedArrList

        Prints false 4 ↗ System.out.println(myArrList == clonedArrList);

        Prints true 5 ↗ String Builder myArrVal = myArrList.get(0);
        Prints true 6 ↗ String Builder assignedArrVal = assignedArrList.get(0);
        Prints true 7 ↗ String Builder clonedArrVal = clonedArrList.get(0);
        Prints true 8 ↗ System.out.println(myArrVal == assignedArrVal);
        Prints true 9 ↗ System.out.println(myArrVal == clonedArrVal);
    }
}

```

Let's go through the previous code:

- ① assigns the object referred to by `myArrList` to `assignedArrList`. The variables `myArrList` and `assignedArrList` now refer to the same object.
- ② assigns a *copy* of the object referred to by `myArrList` to `clonedArrList`. The variables `myArrList` and `clonedArrList` refer to different objects. Because method `clone` returns a value of the type `Object`, it is cast to `ArrayList<StringBuilder>` to assign it to `clonedArrList` (don't worry if you can't follow this line—casting is covered in chapter 6).
- ③ prints `true` because `myArrList` and `assignedArrList` refer to the same object.
- ④ prints `false` because `myArrList` and `clonedArrList` refer to separate objects, because the method `clone` creates and returns a new object of `ArrayList` (but with the same list members).
- ⑤ proves that the method `clone` didn't copy the elements of `myArrList`. All the variable references `myArrVal`, `AssignedArrVal`, and `clonedArrVal` refer to the same objects.
- Hence, both ⑥ and ⑦ print `true`.

CREATING AN ARRAY FROM AN `ArrayList`

You can use the method `toArray` to return an array containing all of the elements in an `ArrayList` in sequence from the first to the last element. As mentioned earlier in this chapter (refer to figure 4.27 in section 4.4.1), an `ArrayList` uses a private variable, '`elementData`' (an array), to store its own values. Method `toArray` doesn't return a reference to this array. It creates a new array, copies the elements of the `ArrayList` to it and then returns it.

Now comes the tricky part. No references to the returned array, which is itself an object, are maintained by the `ArrayList`. But the references to the individual `ArrayList` elements are copied to the returned array and are still referred to by the `ArrayList`.

This implies that if you modify the returned array by, say, swapping the position of its elements or by assigning new objects to its elements, the elements of `ArrayList` won't be affected. But, if you modify the state of (mutable) elements of the returned array, then the modified state of elements will be reflected in the `ArrayList`.

4.5 Comparing objects for equality



[3.3] Test equality between strings and other objects using `==` and `equals()`

In section 4.1, you saw how the class `String` defined a set of rules to determine whether two `String` values are equal, and how these rules were coded in the method `equals`. Similarly, any Java class can define a set of rules to determine whether its two objects should be considered equal. This comparison is accomplished using the method `equals`, which is described in the next section.

4.5.1 The method `equals` in the class `java.lang.Object`

The method `equals` is defined in class `java.lang.Object`. All of the Java classes directly or indirectly inherit from this class. Listing 4.2 contains the default implementation of the method `equals` from class `java.lang.Object`.

Listing 4.2 Implementation of `equals` method from the class `java.lang.Object`

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

As you can see, the default implementation of the `equals` method only compares whether two object variables refer to the same object. Because instance variables are used to store the state of an object, it's common to compare the values of the instance variables to determine whether two objects should be considered equal.

4.5.2 Comparing objects of a user-defined class

Let's work with an example of class `BankAccount`, which defines two instance variables: `acctNumber`, of type `String`, and `acctType`, of type `int`. The `equals` method compares the values of these instance variables to determine the equality of two objects of class `BankAccount`.

Here's the relevant code:

```
class BankAccount {
    String acctNumber;
    int acctType;

    public boolean equals(Object anObject) {
        if (anObject instanceof BankAccount) {
```

Check whether we are comparing the same type of objects

```

        BankAccount b = (BankAccount)anObject;
        return (acctNumber.equals(b.acctNumber) && ←
               acctType == b.acctType);
    }
    else
        return false;
}
}

```

Two bank objects are considered equal if they have the same values, for instance variables acctNumber and acctType

Let's verify the working of this equals method in the following code:

```

class Test {
    public static void main(String args[]) {
        BankAccount b1 = new BankAccount();
        b1.acctNumber = "0023490";
        b1.acctType = 4;

        BankAccount b2 = new BankAccount();
        b2.acctNumber = "11223344";
        b2.acctType = 3;

        BankAccount b3 = new BankAccount();
        b3.acctNumber = "11223344";
        b3.acctType = 3;

        System.out.println(b1.equals(b2));
        System.out.println(b2.equals(b3));
        System.out.println(b1.equals(new String("abc")));
    }
}

```

- ➊ prints false because the value of the reference variables b1 and b2 do not match.
- ➋ prints true because the values of the reference variables b2 and b3 match each other.
- ➌ passes an object of type String to the method equals defined in the class BankAccount. This method returns false if the method parameter passed to it is not of type BankAccount. Hence ➌ prints false.

Even though the following implementation is unacceptable for classes used in the real world, it's still correct syntactically:

```

class BankAccount {
    String acctNumber;
    int acctType;

    public boolean equals(Object anObject) {
        return true;
    }
}

```

The previous definition of the equals method will return true for any object that's compared to an object of class BankAccount because it doesn't compare any values. Let's see what happens when you compare an object of class String with an object of class BankAccount and vice versa using equals():

```

class TestBank {
    public static void main(String args[]) {

```

```

BankAccount acct = new BankAccount();
String str = "Bank";
System.out.println(acct.equals(str));
System.out.println(str.equals(acct));
}
}

```

In the preceding code, ① prints true, but ② prints false. The equals method in the class String returns true only if the object that's being compared to is a String with the same sequence of characters.



EXAM TIP In the exam, watch out for questions about the correct implementation of the equals method to compare two objects versus questions about the equals methods that simply compile correctly. If you'd been asked whether equals() in the previous example code would compile correctly, the correct answer is yes.

4.5.3 Incorrect method signature of the equals method

It's a very common mistake to write an equals method that accepts a member of the class itself. In the following code, the only change is in the type of method parameter:

```

class BankAccount {
    String acctNumber;
    int acctType;

    public boolean equals(BankAccount anObject) {
        if (anObject instanceof BankAccount) {
            BankAccount b = (BankAccount)anObject;
            return (acctNumber.equals(b.acctNumber) &&
                    acctType == b.acctType);
        }
        else
            return false;
    }
}

```

Though the previous definition of equals() may seem to be flawless, what happens when you try to add and retrieve an object of class BankAccount (as shown in the preceding code) from an ArrayList? The method contains defined in the class ArrayList compares two objects by calling the object's equals method. It does not compare object references.

In the following code, see what happens when you add an object of the class BankAccount to an ArrayList and then try to verify whether the list contains a BankAccount object with the same instance variables values for acctNumber and acctType as the object being searched for:

```

class TestMethodEquals {
    public static void main(String args[]) {
        BankAccount b1 = new BankAccount();
        b1.acctNumber = "0023490"; b1.acctType = 4;
}

```

```

ArrayList <BankAccount> list = new ArrayList<BankAccount>();
list.add(b1);

BankAccount b2 = new BankAccount();
b2.acctNumber = "0023490"; b2.acctType = 4;

System.out.println(list.contains(b2));
}
}

① Adds object b1 to list → ②
③ Creates b2 with same state as b1 ← ④ Prints false

```

① and ③ define objects b1 and b2 of the class BankAccount with the same state. ② adds b1 to the list. ④ compares the object b2 with the objects added to the list.

An ArrayList uses the method equals to compare two objects. Because the class BankAccount didn't follow the rules for correctly defining (overriding) the method equals, ArrayList uses the method equals from the base class Object, which compares object references. Because the code didn't add b2 to list, it prints false.

What do you think will be the output of the previous code if you change the definition of the method equals in the class BankAccount so that it accepts a method parameter of type Object? Try it for yourself!



EXAM TIP The method equals defines a method parameter of type Object, and its return type is boolean. Don't change the name of the method, its return type, or the type of method parameter when you define (*override*) this method in your class to compare two objects.

4.5.4 Contract of the equals method

The Java API defines a *contract* for the equals method, which should be taken care of when you implement it in any of your classes. I've pulled the following contract explanation directly from the Java API documentation:¹

The equals method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value x, x.equals(x) should return true.
- It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals() comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

¹ The Java API documentation for equals can be found on the Oracle site: [http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals\(java.lang.Object\)](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)).

As per the contract, the definition of the `equals` method that we defined for the class `BankAccount` in an earlier example violates the contract for the `equals` method. Take a look at the definition again:

```
public boolean equals(Object anObject) {  
    return true;  
}
```

This code returns `true`, even for `null` values passed to this method. According to the contract of the method `equals`, if a `null` value is passed to the `equals` method, the method should return `false`.



EXAM TIP You may get to answer explicit questions on the contract of the `equals` method. An `equals` method that returns `true` for a `null` object passed to it will violate the contract. Also, if the `equals` method modifies the value of any of the instance variables of the method parameter passed to it, or of the object on which it is called, it will violate the `equals` contract.

The `hashCode()` method

A lot of programmers are confused about the role of the method `hashCode` in determining the equality of objects. The method `hashCode` is *not* called by the `equals` method to determine the equality of two objects. Because the `hashCode` method is not on the exam, I'll discuss it quickly here to ward off any confusion about this method.

The method `hashCode` is used by the collection classes (such as `TreeMap` and `HashMap`) that store *key-value* pairs, where a *key* is an object. These collection classes use the `hashCode` of a *key* to search efficiently for the corresponding *value*. The `hashCode` of the *key* (an object) is used to specify a *bucket* number, which should store its corresponding *value*. The `hashCode` values of two objects can be the same. When these collection classes find the right bucket, they call the `equals` method to select the correct *value* object (that shares the same *key* values). The `equals` method is called even if a *bucket* contains only one object. After all, it might be the same hash but a different `equals`, and there is no match to get!

According to the Java documentation, when you override the `equals` method in your class, you should also override the `hashCode` method. If you don't, objects of your classes won't behave as required if they're used as *keys* by collection classes that store *key-value* pairs. This method is not discussed in detail in this chapter because it isn't on the exam. But don't forget to override it with the method `equals` in your real-world projects.

4.6 Summary

In this chapter, you learned about the `String` class, its properties, and its methods. Because this is one of the most frequently used classes in Java, I'll reiterate that a good understanding of this class in terms of why its methods behave in a particular manner will go a long way to helping you successfully complete the OCA Java SE 7 Programmer I exam.

You also learned how to create objects of the class `String` using the operator `new` and the assignment operator (`=`). You also learned the differences between how `String` objects are stored using these two approaches. If you use the assignment operator to create your `String` objects, they're stored in a common pool of `String` objects (also known as the `String` constant pool) that can be used by others. This storage is possible because `String` objects are immutable—that is, their values can't be changed.

You also learned how a `char` array is used to store the value of a `String` object. This knowledge helps explain why the methods `charAt()`, `indexOf()`, and `substring()` search for the first character of a `String` at position 0, not position 1. We also reviewed the methods `replace()`, `trim()`, and `substring()`, which seem to modify the value of a `String` but will never be able to do it because `String` objects are immutable. You also learned the methods `length()`, `startsWith()`, and `endsWith()`.

Because all operators can't be used with `Strings`, you learned about the ones that can be used with `String`: `+`, `+=`, `==`, and `!=`. You also learned that the equality of `Strings` can be determined using the method `equals`. By using the operator `==`, you can only determine whether both the variables are referring to the same object; it doesn't compare the values stored by `Strings`. As with all the other object types, the only literal value that can be assigned to a `String` is `null`.

We worked with the class `StringBuilder`, which is defined in the package `java.lang` and is used to store a mutable sequence of characters. The class `StringBuilder` is usually used to store a sequence of characters that needs to be modified often—like when you're building a query for database applications. Like the `String` class, `StringBuilder` also uses a `char` array to store its characters. A lot of the methods defined in class `StringBuilder` work exactly as defined by the class `String`, such as the methods `charAt`, `indexOf`, `substring`, and `length`. The `append` method is used to add characters to the end of a `StringBuilder` object. The `insert` method is another important `StringBuilder` method that's used to insert either single or multiple characters at a specified position in a `StringBuilder` object. The class `StringBuilder` offers the same functionality offered by the class `StringBuffer`, minus the additional feature of methods that are synchronized where needed.

An array is an object that stores a collection of values. An array can store a collection of primitive data types or a collection of objects. You can define one-dimensional and multidimensional arrays. A one-dimensional array is an object that refers to a collection of scalar values. A two-dimensional (or more) array is referred to as a multidimensional array. A two-dimensional array refers to a collection of objects, where each of the objects is a one-dimensional array. Similarly, a three-dimensional array refers to a collection of two-dimension arrays, and so on. Arrays can be declared, allocated, and initialized in a single step or in multiple steps. A two-dimensional array does not need to be symmetrical, and each of its rows can define different numbers of members. You can define arrays of primitives, interfaces, abstract classes, and concrete classes. All arrays are objects and can access the variable `length` and methods inherited from the class `java.lang.Object`.

`ArrayList` is a resizable array that offers the best combination of features offered by an array and the `List` data structure. You can add objects to an `ArrayList` using the method `add`. You can access the objects of an `ArrayList` by using an enhanced for loop. An `ArrayList` preserves the order of insertion of its elements. `ListIterator` and the enhanced for loop will return the elements in the order in which they were added to the `ArrayList`. You can modify the elements of an `ArrayList` using the method `set`. You can remove the elements of an `ArrayList` by using the method `remove`, which accepts the element position or an object. You can also add multiple elements to an `ArrayList` by using the method `addAll`. The method `clone` defined in the class `ArrayList` returns a *shallow copy* of this `ArrayList` instance. “Shallow copy” means that the method creates a new instance of the `ArrayList` to be cloned, but the `ArrayList` elements aren’t copied.

You can compare the objects of your class by overriding the `equals` method. The `equals` method is defined in the class `java.lang.Object`, which is the base class of all classes in Java. The default implementation of the method `equals` only compares the object references for equality. Because instance variables are used to store the state of an object, it’s common to compare the values of these variables to determine whether two objects should be considered equal in the `equals` method. The Java API documentation defines a contract for the `equals` method. In the exam, for a given definition of the method `equals`, it is important to note the differences between an `equals` method that compiles successfully, one that fails compilation, and one that doesn’t follow the contract.

4.7 Review notes

This section lists the main points covered in this chapter.

The class `String`:

- The class `String` represents an immutable sequence of characters.
- A `String` object can be created by using the operator `new`, by using the assignment operator (`=`), or by using double quotes (as in `System.out.println("Four")`).
- `String` objects created using the assignment operator are placed in a *pool* of `String` objects. Whenever the JRE receives a new request to create a `String` object using the assignment operator, it checks whether a `String` object with the same value already exists in the pool. If found, it returns the object reference for the existing `String` object from the pool.
- `String` objects created using the operator `new` are never placed in the pool of `String` objects.
- The comparison operator (`==`) compares `String` references, whereas the `equals` method compares the `String` values.
- None of the methods defined in the class `String` can modify its value.
- The method `charAt(int index)` retrieves a character at a specified index of a `String`.

- The method `indexOf` can be used to search a `String` for the occurrence of a `char` or a `String`, starting from the first position or a specified position.
- The method `substring` can be used to retrieve a portion of a `String` object. The `substring` method doesn't include the character at the end position.
- The `trim` method will return a new `String` by removing all the leading and trailing white spaces in a `String`. This method doesn't remove any white space *within* a `String`.
- You can use the method `length` to retrieve the length of a `String`.
- The method `startsWith` determines whether a `String` starts with a specified `String`.
- The method `endsWith` determines whether a `String` ends with a specified `String`.
- It's a common practice to use multiple `String` methods in a single line of code. When chained, the methods are evaluated from left to right.
- You can use the concatenation operators `+` and `+=` and comparison operators `!=` and `==` with `String` objects.
- The Java language provides special support for concatenating `String` objects by using the operators `+` and `+=`.
- The right technique for comparing two `String` values for equality is to use the method `equals` defined in the `String` class. This method returns a `true` value if the object being compared isn't `null` and is a `String` object that represents the same sequence of characters as the object to which it's being compared.
- The comparison operator `==` determines whether both the reference variables are referring to the same `String` objects. Hence, it's not the right operator for comparing `String` values.

The class `StringBuilder`:

- The class `StringBuilder` is defined in the package `java.lang` and represents a mutable sequence of characters.
- The `StringBuilder` class is very efficient when a user needs to modify a sequence of characters often. Because it's mutable, the value of a `StringBuilder` object can be modified without the need to create a new `StringBuilder` object.
- A `StringBuilder` object can be created using its constructors, which can accept either a `String` object, another `StringBuilder` object, an `int` value to specify the capacity of `StringBuilder`, or nothing.
- The methods `charAt`, `indexOf`, `substring`, and `length` defined in the class `StringBuilder` work in the same way as methods with the same names defined in the class `String`.
- The `append` method adds the specified value at the end of the existing value of a `StringBuilder` object.
- The `insert` method enables you to insert characters at a specified position in a `StringBuilder` object. The main difference between the `append` and `insert`

methods is that the `insert` method enables you to insert the requested data at a particular position, whereas the `append` method only allows you to add the requested data at the end of the `StringBuilder` object.

- The method `delete` removes the characters in a substring of the specified `StringBuilder`. The method `deleteCharAt` removes the `char` at the specified position.
- Unlike the class `String`, the class `StringBuilder` doesn't define the method `trim`.
- The method `reverse` reverses the sequence of characters of a `StringBuilder`.
- The `replace` method in the class `StringBuilder` replaces a sequence of characters, identified by their position, with another `String`.
- In addition to using the method `substring`, you can also use the method `subSequence` to retrieve a subsequence of a `StringBuilder` object.

Arrays:

- An array is an object that stores a collection of values.
- An array itself is an object.
- An array can store two types of data—a collection of primitive data types and a collection of objects.
- You can define one-dimensional and multidimensional arrays.
- A one-dimensional array is an object that refers to a collection of scalar values.
- A two-dimensional (or more) array is referred to as a multidimensional array.
- A two-dimensional array refers to a collection of objects, in which each of the objects is a one-dimensional array.
- Similarly, a three-dimensional array refers to a collection of two-dimensional arrays, and so on.
- Multidimensional arrays may or may not contain the same number of elements in each row or column.
- The creation of an array involves three steps: declaration of an array, allocation of an array, and initialization of array elements.
- An array declaration is composed of an array type, a variable name, and one or more occurrences of `[]`.
- Square brackets can follow either the variable name or its type. In the case of multidimensional arrays, it can follow both of them.
- An array declaration creates a variable that refers to `null`.
- Because no elements of an array are created when it's declared, it's invalid to define the size of an array with its declaration.
- *Array allocation* allocates memory for the elements of an array. When you allocate memory for an array, you must specify its dimensions, such as the number of elements the array should store.
- Because an array is an object, it's allocated using the keyword `new`, followed by the type of value that it stores, and then its size.

- Once allocated, all the array elements store their default values. Elements of an array that store objects refer to null. Elements of an array that store primitive data types store 0 for integer types (byte, short, int, long), 0.0 for decimal types (float and double), false for boolean, or \u0000 for char data.
- To access an element in a two-dimensional array, use two array position values.
- You can combine all the steps of array declaration, allocation, and initialization into one single step.
- When you combine array declaration, allocation, and initialization in a single step, you can't specify the size of the array. The size of the array is calculated by the number of values that are assigned to the array.
- You can declare and allocate an array but choose not to initialize it (for example, `int [] a = new int [5];`).
- The Java compiler doesn't check the range of the index positions at which you try to access an array element. The code throws an `ArrayIndexOutOfBoundsException` exception if the requested index position doesn't fall in the valid range at runtime.
- A multidimensional array can be asymmetrical; it may not define the same number of columns for each of its rows.
- The type of an array can also be an interface or abstract class. Such an array can be used to store objects of classes that inherit from the interface type or the abstract class type.
- The type of an array can also be `java.lang.Object`. Because all classes extend the class `java.lang.Object` class, elements of this array can refer to any object.
- All the arrays are objects and can access the variable `length`, which specifies the number of components stored by the array.
- Because all arrays are objects, they inherit and can access all methods from the class `Object`.

ArrayList:

- `ArrayList` is one of the most widely used classes from the Collections framework. It offers the best combination of features offered by an array and the `List` data structure.
- An `ArrayList` is like a resizable array.
- Unlike arrays, you may not specify an initial size to create an `ArrayList`.
- `ArrayList` implements the interface `List` and allows null values to be added to it.
- `ArrayList` implements all list operations (add, modify, and delete values).
- `ArrayList` allows duplicate values to be added to it and maintains its insertion order.
- You can use either `Iterator` or `ListIterator` to iterate over the items of an `ArrayList`.

- `ArrayList` supports generics, making it type safe.
- Internally, an array of type `java.lang.Object` is used to store the data in an `ArrayList`.
- You can add a value to an `ArrayList` either at its end or at a specified position by using the method `add`.
- To access the elements of an `ArrayList`, you can use either the enhanced for loop, `Iterator`, or `ListIterator`.
- An iterator (`Iterator` or `ListIterator`) lets you remove elements as you iterate through an `ArrayList`. It's not possible to remove elements from an `ArrayList` while iterating through it using a for loop.
- An `ArrayList` preserves the order of insertion of its elements. `ListIterator` and the enhanced for loop will return the elements in the order in which they were added to the `ArrayList`.
- You can use the method `set` to modify an `ArrayList` by either replacing an existing element in `ArrayList` or modifying its existing values.
- `remove(int)` removes the element at the specified position in the list.
- `remove(Object o)` removes the first occurrence of the specified element from the list, if it's present.
- You can add multiple elements to an `ArrayList` from another `ArrayList` or any other class that's a subclass of `Collection` by using the method `addAll`.
- You can remove all the `ArrayList` elements by calling the method `clear` on it.
- `get(int index)` returns the element at the specified position in the list.
- `size()` returns the number of elements in the list.
- `contains(Object o)` returns true if the list contains the specified element.
- `indexOf(Object o)` returns the index of the first occurrence of the specified element in the list, or -1 if the list doesn't contain the element.
- `lastIndexOf(Object o)` returns the index of the last occurrence of the specified element in the list, or -1 if the list doesn't contain the element.
- The method `clone` defined in the class `ArrayList` returns a *shallow copy* of this `ArrayList` instance. "Shallow copy" means that the method creates a new instance of the `ArrayList` to be cloned, but the `ArrayList` elements aren't copied.
- You can use the method `toArray` to return an array containing all of the elements in `ArrayList` in sequence from the first to the last element.

Comparing objects for equality:

- Any Java class can define a set of rules to determine whether two objects should be considered equal.
- The method `equals` is defined in the class `java.lang.Object`. All the Java classes directly or indirectly inherit this class.
- The default implementation of the `equals` method only compares whether two object variables refer to the same object.

- Because instance variables are used to store the state of an object, it's common to compare the values of the instance variables to determine whether two objects should be considered equal.
- When you override the `equals` method in your class, make sure that you use the correct method signature for the `equals` method.
- The Java API defines a contract for the `equals` method, which should be taken care of when you implement the method in any of your classes.
- According to the contract of the method `equals`, if a `null` value is passed to it, the method `equals` should return `false`.
- If the `equals` method modifies the value of any of the instance variables of the method parameter passed to it, or of the object on which it is called, it will violate the contract.

4.8 Sample exam questions

Q4-1. What is the output of the following code?

```
class EJavaGuruArray {
    public static void main(String args[]) {
        int[] arr = new int[5];
        byte b = 4; char c = 'c'; long longVar = 10;
        arr[0] = b;
        arr[1] = c;
        arr[3] = longVar;
        System.out.println(arr[0] + arr[1] + arr[2] + arr[3]);
    }
}

a 4c010
b 4c10
c 113
d 103
e Compilation error
```

Q4-2. What is the output of the following code?

```
class EJavaGuruArray2 {
    public static void main(String args[]) {
        int[] arr1;
        int[] arr2 = new int[3];
        char[] arr3 = {'a', 'b'};
        arr1 = arr2;
        arr1 = arr3;
        System.out.println(arr1[0] + ":" + arr1[1]);
    }
}

a 0:0
b a:b
c 0:b
```

- d** a:0
- e** Compilation error

Q4-3. Which of the following are valid lines of code to define a multidimensional int array?

- a** int [] [] array1 = {{1, 2, 3}, {}, {1, 2, 3, 4, 5}};
- b** int [] [] array2 = new array() {{1, 2, 3}, {}, {1, 2, 3, 4, 5}};
- c** int [] [] array3 = {1, 2, 3}, {0}, {1, 2, 3, 4, 5};
- d** int [] [] array5 = new int [2] [] ;

Q4-4. Which of the following statements are correct?

- a** By default, an ArrayList creates an array with an initial size of 16 to store its elements.
- b** Because ArrayList stores only objects, you can't pass an element of an ArrayList to a switch construct.
- c** Calling clear() and remove() on an ArrayList will remove all its elements.
- d** If you frequently add elements to an ArrayList, specifying a larger capacity will improve the code efficiency.
- e** Calling the method clone() on an ArrayList creates its shallow copy; that is, it doesn't clone the individual list elements.

Q4-5. Which of the following statements are correct?

- a** An ArrayList offers a resizable array, which is easily managed using the methods it provides. You can add and remove elements from an ArrayList.
- b** Values stored by an ArrayList can be modified.
- c** You can iterate through elements of an ArrayList using a for loop, Iterator, or ListIterator.
- d** An ArrayList requires you to specify the total number of elements before you can store any elements in it.
- e** An ArrayList can store any type of object.

Q4-6. What is the output of the following code?

```

import java.util.*;                                     // line 1
class EJavaGuruArrayList {                           // line 2
    public static void main(String args[]) {          // line 3
        ArrayList<String> ejg = new ArrayList<>();   // line 4
        ejg.add("One");                                // line 5
        ejg.add("Two");                                // line 6
        System.out.println(ejg.contains(new String("One"))); // line 7
        System.out.println(ejg.indexOf("Two"));           // line 8
        ejg.clear();                                   // line 9
        System.out.println(ejg);                         // line 10
        System.out.println(ejg.get(1));                  // line 11
    }                                                 // line 12
}                                                 // line 13

```

- a Line 7 prints true
- b Line 7 prints false
- c Line 8 prints -1
- d Line 8 prints 1
- e Line 9 removes all elements of the list ejg
- f Line 9 sets the list ejg to null
- g Line 10 prints null
- h Line 10 prints []
- i Line 10 prints a value similar to ArrayList@16356
- k Line 11 throws an exception
- l Line 11 prints null

Q4-7. What is the output of the following code?

```
class EJavaGuruString {
    public static void main(String args[]) {
        String ejg1 = new String("E Java");
        String ejg2 = new String("E Java");
        String ejg3 = "E Java";
        String ejg4 = "E Java";
        do
            System.out.println(ejg1.equals(ejg2));
        while (ejg3 == ejg4);
    }
}
```

- a true printed once
- b false printed once
- c true printed in an infinite loop
- d false printed in an infinite loop

Q4-8. What is the output of the following code?

```
class EJavaGuruString2 {
    public static void main(String args[]) {
        String ejg = "game".replace('a', 'Z').trim().concat("Aa");
        ejg.substring(0, 2);
        System.out.println(ejg);
    }
}



- a gZmeAZ
- b gZmeAa
- c gZm
- d gZ
- e game

```

Q4-9. What is the output of the following code?

```
class EJavaGuruString2 {  
    public static void main(String args[]) {  
        String ejg = "game";  
        ejg.replace('a', 'Z').trim().concat("Aa");  
        ejg.substring(0, 2);  
        System.out.println(ejg);  
    }  
}  
  
a gZmeAZ  
b gZmeAa  
c gZm  
d gZ  
e game
```

Q4-10. What is the output of the following code?

```
class EJavaGuruStringBuilder {  
    public static void main(String args[]) {  
        StringBuilder ejg = new StringBuilder(10 + 2 + "SUN" + 4 + 5);  
        ejg.append(ejg.delete(3, 6));  
        System.out.println(ejg);  
    }  
}  
  
a 12S512S5  
b 12S12S  
c 1025102S  
d Runtime exception
```

Q4-11. What is the output of the following code?

```
class EJavaGuruStringBuilder2 {  
    public static void main(String args[]) {  
        StringBuilder sb1 = new StringBuilder("123456");  
        sb1.subSequence(2, 4);  
        sb1.deleteCharAt(3);  
        sb1.reverse();  
        System.out.println(sb1);  
    }  
}  
  
a 521  
b Runtime exception  
c 65321  
d 65431
```

4.9 Answers to sample exam questions

Q4-1. What is the output of the following code?

```
class EJavaGuruArray {
    public static void main(String args[]) {
        int[] arr = new int[5];
        byte b = 4; char c = 'c'; long longVar = 10;
        arr[0] = b;
        arr[1] = c;
        arr[3] = longVar;
        System.out.println(arr[0] + arr[1] + arr[2] + arr[3]);
    }
}

a 4c010
b 4c10
c 113
d 103
e Compilation error
```

Answer: e

Explanation: The previous code won't compile due to the following line of code:

```
arr[3] = longVar;
```

This line of code tries to assign a value of type `long` to a variable of type `int`. Because Java does support implicit widening conversions for variables, the previous code fails to compile. Also, the previous code tries to trick you regarding your understanding of the following:

- Assigning a `char` value to an `int` array element (`arr[1] = c`)
- Adding a `byte` value to an `int` array element (`arr[0] = b`)
- Whether an unassigned `int` array element is assigned a default value (`arr[2]`)
- Whether `arr[0] + arr[1] + arr[2] + arr[3]` prints the sum of all these values, or a concatenated value

When answering questions in the OCA Java SE 7 Java Programmer I exam, be careful about such tactics. If any of the answers list a compilation error or a runtime exception as an option, look for obvious lines of code that could result in it. In this example, `arr[3] = longVar` will result in compilation error.

Q4-2. What is the output of the following code?

```
class EJavaGuruArray2 {
    public static void main(String args[]) {
        int[] arr1;
        int[] arr2 = new int[3];
        char[] arr3 = {'a', 'b'};
        arr1 = arr2;
        arr1 = arr3;
    }
}
```

```

        System.out.println(arr1[0] + ":" + arr1[1]);
    }
}

a 0:0
b a:b
c 0:b
d a:0
e Compilation error

```

Answer: e

Explanation: Because a char value can be assigned to an int value, you might assume that a char array can be assigned to an int array. But we're talking about arrays of int and char primitives, which aren't the same as a primitive int or char. Arrays themselves are reference variables, which refer to a collection of objects of similar type.

Q4-3. Which of the following are valid lines of code to define a multidimensional int array?

- a `int[][] array1 = {{1, 2, 3}, {}, {1, 2, 3, 4, 5}};`
- b `int[][] array2 = new array() {{1, 2, 3}, {}, {1, 2, 3, 4, 5}};`
- c `int[][] array3 = {1, 2, 3}, {0}, {1, 2, 3, 4, 5};`
- d `int[][] array5 = new int[2][];`

Answer: a, d

Explanation: Option (b) is incorrect. This line of code won't compile because `new array()` isn't valid code. Unlike objects of other classes, an array isn't initialized using the keyword `new` followed by the word `array`. When the keyword `new` is used to initialize an array, it's followed by the type of the array, not the word `array`.

Option (c) is incorrect. To initialize a two-dimensional array, all of these values must be enclosed within another pair of curly braces, as shown in option (a).

Q4-4. Which of the following statements are correct?

- a By default, an `ArrayList` creates an array with an initial size of 16 to store its elements.
- b Because `ArrayList` stores only objects, you can't pass element of an `ArrayList` to a `switch` construct.
- c Calling `clear()` or `remove()` on an `ArrayList`, will remove all its elements.
- d If you frequently add elements to an `ArrayList`, specifying a larger capacity will improve the code efficiency.
- e Calling the method `clone()` on an `ArrayList` creates its shallow copy; that is, it doesn't clone the individual list elements.

Answer: d, e

Explanation: Option (a) is incorrect. By default, an `ArrayList` creates an array with an initial size of 10 to store its elements.

Option (b) is incorrect. Starting with Java 7, switch also accepts variables of type String. Because a String can be stored in an ArrayList, you can use elements of an ArrayList in a switch construct.

Option (c) is incorrect. Only remove() will remove all elements of an ArrayList.

Option (d) is correct. An ArrayList internally uses an array to store all its elements. Whenever you add an element to an ArrayList, it checks whether the array can accommodate the new value. If it can't, ArrayList creates a larger array, copies all the existing values to the new array, and then adds the new value at the end of the array. If you frequently add elements to an ArrayList, it makes sense to create an ArrayList with a bigger capacity because the previous process isn't repeated for each ArrayList insertion.

Option (e) is correct. Calling clone() on an ArrayList will create a separate reference variable that stores the same number of elements as the ArrayList to be cloned. But each individual ArrayList element will refer to the same object; that is, the individual ArrayList elements aren't cloned.

Q4-5. Which of the following statements are correct?

- a **An ArrayList offers a resizable array, which is easily managed using the methods it provides. You can add and remove elements from an ArrayList.**
- b **Values stored by an ArrayList can be modified.**
- c **You can iterate through elements of an ArrayList using a for loop, Iterator, or ListIterator.**
- d An ArrayList requires you to specify the total elements before you can store any elements in it.
- e **An ArrayList can store any type of object.**

Answer: a, b, c, e

Explanation: Option (a) is correct. A developer may prefer using an ArrayList over an array because it offers all the benefits of an array and a list. For example, you can easily add or remove elements from an ArrayList.

Option (b) is correct.

Option (c) is correct. An ArrayList can be easily searched, sorted, and have its values compared using the methods provided by the Collection framework classes.

Option (d) is incorrect. An array requires you to specify the total number of elements before you can add any element to it. But you don't need to specify the total number of elements that you may add to an ArrayList at any time in your code.

Option (e) is correct.

Q4-6. What is the output of the following code?

```
import java.util.*; // line 1
class EJavaGuruArrayList { // line 2
    public static void main(String args[]) { // line 3
        ArrayList<String> ejg = new ArrayList<>(); // line 4
```

```

ejg.add("One");                                // line 5
ejg.add("Two");                                // line 6

System.out.println(ejg.contains(new String("One"))); // line 7
System.out.println(ejg.indexOf("Two"));           // line 8
ejg.clear();                                    // line 9
System.out.println(ejg);                         // line 10
System.out.println(ejg.get(1));                  // line 11

}
}                                                 // line 12
// line 13

```

- a Line 7 prints true**
- b Line 7 prints false**
- c Line 8 prints -1**
- d Line 8 prints 1**
- e Line 9 removes all elements of the list ejg**
- f Line 9 sets ejg to null**
- g Line 10 prints null**
- h Line 10 prints []**
- i Line 10 prints a value similar to ArrayList@16356**
- k Line 11 throws an exception**
- l Line 11 prints null**

Answer: a, d, e, h, k

Explanation: Line 7: The method `contains` accepts an object and compares it with the values stored in the list. It returns `true` if the method finds a match and `false` otherwise. This method uses the `equals` method defined by the object stored in the list. In the example, the `ArrayList` stores objects of class `String`, which has overridden the `equals` method. The `equals` method of the `String` class compares the values stored by it. This is why line 7 returns the value `true`.

Line 8: `indexOf` returns the index position of an element if a match is found; otherwise, it returns `-1`. This method also uses the `equals` method behind the scenes to compare the values in an `ArrayList`. Because the `equals` method in class `String` compares its values and not the reference variables, the `indexOf` method finds a match in position 1.

Line 9: The `clear` method removes all the individual elements of an `ArrayList` such that an attempt to access any of the earlier `ArrayList` elements will throw a runtime exception. It doesn't set the `ArrayList` reference variable to `null`.

Line 10: `ArrayList` has overridden the `toString` method such that it returns a list of all its elements enclosed within square brackets. To print each element, the `toString` method is called to retrieve its `String` representation.

Line 11: The `get` method removes all the elements of an `ArrayList`. An attempt to access the (nonexistent) `ArrayList` element throws a runtime `IndexOutOfBoundsException` exception.

This question tests your understanding of `ArrayList` and determining the equality of `String` objects.

Q4-7. What is the output of the following code?

```
class EJavaGuruString {
    public static void main(String args[]) {
        String ejg1 = new String("E Java");
        String ejg2 = new String("E Java");
        String ejg3 = "E Java";
        String ejg4 = "E Java";
        do
            System.out.println(ejg1.equals(ejg2));
            while (ejg3 == ejg4);
    }
}
```

- a true printed once
- b false printed once
- c **true printed in an infinite loop**
- d false printed in an infinite loop

Answer: c

Explanation: `String` objects that are created without using the `new` operator are placed in a pool of `Strings`. Hence, the `String` object referred to by the variable `ejg3` is placed in a pool of `Strings`. The variable `ejg4` is also defined without using the `new` operator. Before Java creates another `String` object in the `String` pool for the variable `ejg4`, it looks for a `String` object with the same value in the pool. Because this value already exists in the pool, it makes the variable `ejg4` refer to the same `String` object. This, in turn, makes the variables `ejg3` and `ejg4` refer to the same `String` objects. Hence, both of the following comparisons will return `true`:

- `ejg3 == ejg4` (compare the object references)
- `ejg3.equals(ejg4)` (compare the object values)

Even though the variables `ejg1` and `ejg2` refer to different `String` objects, they define the same values. So `ejg1.equals(ejg2)` also returns `true`. Because the loop condition (`ejg3==ejg4`) always returns `true`, the code prints `true` in an infinite loop.

Q4-8. What is the output of the following code?

```
class EJavaGuruString2 {
    public static void main(String args[]) {
        String ejg = "game".replace('a', 'Z').trim().concat("Aa");
        ejg.substring(0, 2);
        System.out.println(ejg);
    }
}
```

- a `gZmeAZ`
- b `gZmeAa`

- c** gZm
- d** gZ
- e** game

Answer: b

Explanation: When chained, methods are evaluated from left to right. The first method to execute is replace, not concat. Strings are immutable. Calling the method substring on the reference variable ejg doesn't change the contents of the variable ejg. It returns a String object that isn't referred to by any other variable in the code. In fact, none of the methods defined in the String class modifies the object's own value. They all create and return new String objects.

Q4-9. What is the output of the following code?

```
class EJavaGuruString2 {
    public static void main(String args[]) {
        String ejg = "game";
        ejg.replace('a', 'Z').trim().concat("Aa");
        ejg.substring(0, 2);
        System.out.println(ejg);
    }
}

a gZmeAZ
b gZmeAa
c gZm
d gZ
e game
```

Answer: e

Explanation: String objects are immutable. It doesn't matter how many methods you execute on a String object; its value won't change. Variable ejg is initialized with the String value "game". This value won't change, and the code prints game.

Q4-10. What is the output of the following code?

```
class EJavaGuruStringBuilder {
    public static void main(String args[]) {
        StringBuilder ejg = new StringBuilder(10 + 2 + "SUN" + 4 + 5);
        ejg.append(ejg.delete(3, 6));
        System.out.println(ejg);
    }
}

a 12S512S5
b 12S12S
c 1025102S
d dRuntime exception
```

Answer: a

Explanation: This question tests you on your understanding of operators, `String`, and `StringBuilder`. The following line of code returns `12SUN45`:

```
10 + 2 + "SUN" + 4 + 5
```

The `+` operator adds two numbers but concatenates the last two numbers. When the `+` operator encounters a `String` object, it treats all the remaining operands as `String` objects.

Unlike the `String` objects, `StringBuilder` objects are mutable. The `append` and `delete` methods defined in this class change its value. `eJg.delete(3, 6)` modifies the existing value of the `StringBuilder` to `12S5`. It then appends the same value to itself when calling `eJg.append()`, resulting in the value `12S512S5`.

Q4-11. What is the output of the following code?

```
class EJavaGuruStringBuilder2 {  
    public static void main(String args[]) {  
        StringBuilder sb1 = new StringBuilder("123456");  
        sb1.subSequence(2, 4);  
        sb1.deleteCharAt(3);  
        sb1.reverse();  
        System.out.println(sb1);  
    }  
}  
  
a 521  
b Runtime exception  
c 65321  
d 65431
```

Answer: c

Explanation: Like the method `substring`, the method `subSequence` doesn't modify the contents of a `StringBuilder`. Hence, the value of the variable `sb1` remains `123456`, even after the execution of the following line of code:

```
sb1.subSequence(2, 4);
```

The method `deleteCharAt` deletes a `char` value at position 3. Because the positions are zero-based, the digit 4 is deleted from the value `123456`, resulting in `12356`. The method `reverse` modifies the value of a `StringBuilder` by assigning to it the reverse representation of its value. The reverse of `12356` is `65321`.