# 3

# Programming with Java Operators and Strings

## CERTIFICATION OBJECTIVES

- Understanding Fundamental Operators
- Developing with String Objects and Their Methods

✓ Two-Minute Drill

Q&A Self Test

Two of the most fundamental elements of the Java programming language are Java operators and strings. This chapter discusses Java operators and how they manipulate their operands. You will need a full understanding of the different types and groupings of operators to score well on the exam. This chapter provides you with all of the operator-related information you need to know.

Strings are commonly used in Java, therefore they will also be present throughout the exam. This chapter details the `String` class and its related functionality. Topics include the string concatenation operator and the `toString` method, as well as a discussion of valuable methods straight from the `String` class.

After completing this chapter, you will have all the knowledge necessary to score well on the operator- and string-related questions on the exam.

## CERTIFICATION OBJECTIVE

# Understanding Fundamental Operators

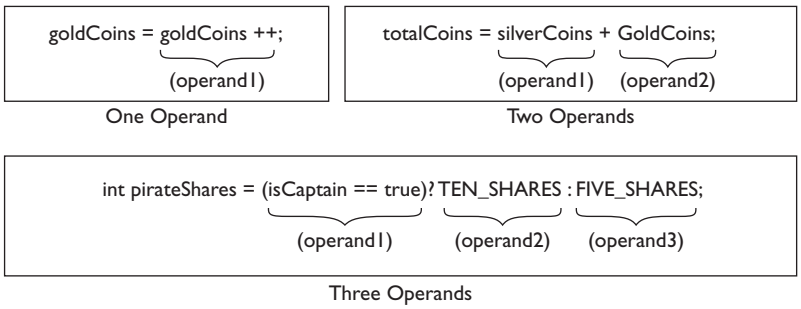
*Exam Objective 4.5 Given an algorithm as pseudo-code, develop code that correctly applies the appropriate operators, including assignment operators (limited to: =, +=, -=), arithmetic operators (limited to: +, -, *, /, %, ++, --), relational operators (limited to: <, <=, >, >=, ==, !=), logical operators (limited to: !, &&, ||), to produce a desired result. Also, write code that determines the equality of two objects or two primitives.*

Java operators are used to return a result from an expression using one, two, or three operands. Operands are the values placed to the right or left side of the operators. Prefix/postfix-increment/decrement operators use one operand. The conditional ternary operator (`?:`) uses three operands. All other operators use two operands. Examples of "operand use" are shown in Figure 3-1. Note that the result of evaluating operands is typically a primitive value.

Table 3-1 represents all of the operators you may see on the exam. The precedence defines the order of which operator will be evaluated when several are included in an expression. The association defines which operand will be used (or evaluated) first. The Java operators that are not on the exam are the bitwise complement (`~`), left shift (`<<`), right shift (`>>`), unsigned right shift (`>>>`), boolean AND (`&`), bitwise AND (`&`), boolean exclusive OR (`^`), bitwise OR(`^`), boolean OR (`|`), bitwise OR(`|`), conditional ternary operator (`?:`), and the following compound assignment operators (`*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`).

**FIGURE 3-1**

Operands




The following topics will be covered in these pages:

- Assignment operators
- Arithmetic operators
- Relational operators
- Logical operators

**TABLE 3-1** Java Operators on the SCJA Exam

| Relative Precedence | Operator | Description | Operator Type | Association |
|---|---|---|---|---|
| 1 | ++,- - | Postfix increment, postfix decrement | Arithmetic | Right to left |
| 2 | ++,- - | Prefix increment, prefix decrement | Arithmetic | Right to left |
| 2 | ! | Boolean NOT | Logical | Right to left |
| 3 | *,/,% | Multiplication, division, remainder (modulus) | Arithmetic | Left to right |
| 4 | +,- | Addition, subtraction | Arithmetic | Left to right |
| 5 | <, <=, >, >= | Less than, less than or equal to, greater than, greater than or equal to | Relational | Left to right |
| 6 | ==, != | Value equality and inequality | Relational | Left to right |
| 6 | ==, != | Reference equality and inequality | Relational | Left to right |
| 7 | && | Conditional AND | Logical | Left to right |
| 8 | \|\| | Conditional OR | Logical | Left to right |
| 9 | =, +=, -= | Assignment and compound assignments (addition and subtraction) | Assignment | Right to left |

## Assignment Operators

Assignment operators are used to assign values to variables.

- = assignment operator

The assignment operator by itself is the equal sign (=). Chapter 2 discusses assignment statements, and Chapter 4 discusses the assignment of literals into primitive data types and the creation of reference type variables. At its simplest, the assignment operators move valid literals into variables or cause compiler errors when the literals are not valid. The following are valid assignment statements using the assignment operator.

```
boolean hasTreasureChestKey = true;
byte shipmates = 20;
PirateShip ship = new PirateShip();
```

The following are invalid assignments and will cause compiler errors:

```
/* Invalid literal, TRUE must be lower case */
boolean hasTreasureChestKey = TRUE;
/* Invalid literal, byte value cannot exceed 127 */
byte shipmates = 500;
/*  Invalid constructor */
PirateShip ship = new PirateShip(UNEXPECTED_ARG);
```

### Compound Assignment Operators

A variety of compound assignment operators exist. The exam only covers the addition and subtraction compound assignment operators.

- += assignment by addition operator
- -= assignment by subtraction operator

Consider the following two assignment statements:

```
goldCoins = goldCoins + 99;
pirateShips = pirateShips - 1;
```

The following two statements (with the same meaning and results as earlier) are written with compound assignment operators.

```
goldCoins += 99;
pirateShips -= 1;
```

on the job

***While the use of compound assignment operators cuts down on keystrokes, it is generally good practice to use the former "longhand" approach since the code is clearly more readable.***

exam watch

***It is common to represent assignments in pseudo-code with the colon and equal sign characters (for example, A := 20). Notice that := looks strikingly familiar to +=, -=, and other Java assignment operators such as *=, /=, and %=. Be aware though that the pseudo-code assignment representation (:=) is not a Java assignment operator, and if you see it in any Java code, it will not compile.***

## EXERCISE 3-1

### Using Compound Assignment Operators

This exercise will clear up any confusion about compound assignment operators. The following application will be used for the exercise. Don't run it until after step 3.

```
public class Main {
  public static void main(String[] args) {
    byte a;
    a = 10;
    System.out.println(a += 3);
    a = 15;
    System.out.println(a -= 3);
    a = 20;
    System.out.println(a *= 3);
    a = 25;
    System.out.println(a /= 3);
    a = 30;
    System.out.println(a %= 3);
    /* Optional as outside of scope of exam */
    a = 35;
    System.out.println(a &= 3);
    a = 40;
    System.out.println(a ^= 3);
    a = 45;
```

```
        System.out.println(a |= 3);
        a = 50;
        System.out.println(a <<= 3);
        a = 55;
        System.out.println(a >>= 3);
        a = 60;
        System.out.println(a >>>= 3);
        /* End optional */
    }
}
```

1. Grab a pencil and a piece of paper. Optionally, you can use Table 3-2 as your worksheet.
2. For each statement that has a compound assignment operator, rewrite the statement without the compound assignment operator and replace the variable with its associated value. For example, let's take the assignment statement with the addition compound assignment operator:

   ```
   a = 5;
   System.out.println(a += 3);
   ```

   It would be rewritten as `(a = a + 3)`, specifically `(a = 5 + 3);`

**TABLE 3-2** Refactoring Compound Assignment Statements

| Assigned Value of a | Compound Assignment | Refactored Statement | New Value of a |
|---|---|---|---|
| `a = 10;` | `a += 3;` | `a = 10 + 3;` | 13 |
| `a = 15;` | `a -= 3;` | | |
| `a = 20;` | `a *= 3;` | | |
| `a = 25;` | `a /= 3;` | | |
| `a = 30;` | `a %= 3;` | | |
| `a = 35;` | `a &= 3;` | | |
| `a = 40;` | `a ^= 3;` | | |
| `a = 45;` | `a \|= 3;` | | |
| `a = 50;` | `a <<= 3;` | | |
| `a = 55;` | `a >>= 3;` | | |
| `a = 60;` | `a >>>= 3;` | | |

3. Evaluate the expressions, without using a computer.
4. Compile and run the given application. Compare your results.

   Note that many of these operators do not appear on the exam. The point of the exercise is getting you properly acquainted with compound assignment operators, by repetition.

## Arithmetic Operators

The exam will include nine arithmetic operators. Five of these operators are used for basic operations (addition, subtraction, multiplication, division, and modulus). The other four operators are used for incrementing and decrementing a value. We'll examine the five operators used for basic operations first.

### Basic Arithmetic Operators

The five basic arithmetic operators are

- `+` addition (sum) operator
- `-` subtraction (difference) operator
- `*` multiplication (product) operator
- `/` division (quotient) operator
- `%` modulus (remainder) operator

Adding, subtracting, multiplying, dividing, and producing remainders with operators is straightforward. The following examples demonstrate this.

```
/* Addition (+) operator example */
int greyCannonBalls = 50;
int blackCannonBalls = 50;
int totalCannonBalls = greyCannonBalls + blackCannonBalls; // 100

/* Subtraction (-) operator example */
int firedCannonBalls = 10;
totalCannonBalls = totalCannonBalls - firedCannonBalls; // 90

/* Multiplication (*) operator example */
int matches = 20;
```

```
int matchboxes = 20;
int totalMatches = matches * matchboxes; // 400

/* Division (/) operator example */
int pirates = 104;
int pirateShips = 3;
int assignedPiratesPerShip = pirates / pirateShips; // 34

/* Remainder (modulus) (%) operator example */
(left operand is divided by right operand and the remainder is
produced)
int pirateRemainder = pirates % pirateShips; // 2
```

## Prefix Increment, Postfix Increment, Prefix Decrement, and Postfix Decrement Operators

Four operators allow decrementing or incrementing of variables:

- `++x` prefix increment operator
- `--x` prefix decrement operator
- `x++` postfix increment operator
- `x--` postfix decrement operator

Prefix increment and prefix decrement operators provide a shorthand way of incrementing the value of a variable by 1. Rather than creating an expression as `y=x+1`, you could write `y=++x`. Similarly, you could replace the expression `y=x-1` with `y=--x`. This works because the execution of the prefix operators occurs on the operand prior to the evaluation of the whole expression. Postfix increment and postfix decrement characters execute the postfix operators after the expression has been evaluated. Therefore `y = x++` would equate to `y=x` followed by `x=x+1`. And `y = x--` would equate to `y=x`, followed by `x=x-1`.

It's important to note that `y=++x` is not exactly equivalent to `y=x+1` because the value of x changes in the former but not in the latter. This is the same for `y=--x` and `y=x-1`.

The prefix increment operator increments a value by 1 before an expression has been evaluated.

```
int x = 10;
int y = ++x ;
System.out.println("x=" + x + ", y=" + y); // x= 11, y= 11
```

The postfix increment operator increments a value by 1 after an expression has been evaluated.

```
int x = 10;
int y = x++ ;
System.out.println("x=" + x + ", y=" + y); // x= 11, y= 10
```

The prefix decrement operator decrements a value by 1 before an expression has been evaluated.

```
int x = 10;
int y = --x ;
System.out.println("x=" + x + ", y=" + y); // x= 9, y= 9
```

The postfix decrement operator decrements a value by 1 after an expression has been evaluated.

```
int x = 10;
int y = x-- ;
System.out.println("x=" + x + ", y=" + y); // x= 9, y= 10
```

## Relational Operators

Relational operators return Boolean values in relationship to the evaluation of their left and right operands. The six most common relational operators are on the exam. Four of them equate to the greater than and less than comparisons. Two are strictly related to equality as we will discuss at the end of this section.

### Basic Relational Operators

- `<` "less than" operator
- `<=` "less than or equal to" operator
- `>` "greater than" operator
- `>=` "greater than or equal to" operator

The "less than," "less than or equal to," "greater than," and "greater than or equal to" operators are used to compare integers, floating points, and characters. When the expression used with the relational operators is true, the Boolean value of true is returned; otherwise, false is returned.

```
/* returns true as 1 is less than 2 */
boolean b1 = 1 < 2;
/* returns false as 3 is not less than 2 */
boolean b2 = 3 < 2;
```

```
/* returns true as 3 is greater than 2 */
boolean b3 = 3 > 2;
/* returns false as 1 is not greater than 2 */
boolean b4 = 1 > 2;
/* returns true as 2 is less than or equal to 2 */
boolean b5 = 2 <= 2;
/* returns false as 3 is not less than or equal to 2 */
boolean b6 = 3 <= 2;
/* returns true as 3 is greater than or equal to 3 */
boolean b7 = 3 >= 3;
/* returns false as 2 is not greater than or equal to 3 */
boolean b8 = 2 >= 3;
```

So far we've only examined the relationship of `int` primitives. Let's take a look at the various ways `char` primitives can be evaluated with relational operators, specifically the "less than" operator for these examples. Remember that characters (that is, `char` primitives) accept integers (within the valid 16-bit unsigned range), hexadecimal, octal, and character literals. Each literal in the following examples represents the letters "A" and "B." The left operands are character "A" and the right operands are character "B." Since each expression is essentially the same, they all evaluate to true.

```
boolean b1 = 'A' < 'B'; // Character literals
boolean b2 = '\u0041' < '\u0042'; // Unicode literals
boolean b3 = 0x0041 < 0x0042; // Hexadecimal literals
boolean b4 = 65 < 66; // Integer literals that fit in a char
boolean b5 = 0101 < 0102; //Octal literals
boolean b6 = '\101' < '\102'; //Octal literals with escape
sequences
boolean b7 = 'A' < 0102; // Character and Octal literals
```

As mentioned, you can also test the relationship between floating points. The following are a few examples.

```
boolean b1 = 9.00D < 9.50D; // Floating points with D postfixes
boolean b2 = 9.00d < 9.50d; // Floating points with d postfixes
boolean b3 = 9.00F < 9.50F; // Floating points with F postfixes
boolean b4 = 9.0f < 9.50f; // Floating points with f postfixes
boolean b5 = (double)9 < (double)10; // Integers with explicit
casts
boolean b6 = (float)9 < (float)10; // Integers with explicit casts
boolean b7 = 9 < 10; // Integers that fit into floating points
boolean b8 = (9d < 10f);
boolean b9 = (float)11 < 12;
```

## Equality Operators

Relational operators that directly compare the equality of primitives (numbers, characters, Booleans) and object reference variables are considered equality operators.

- `==` "equal to" operator
- `!=` "not equal to" operator

Comparing primitives of the same type is straightforward. If the right and left operands of the "equal to" operator are equal, the Boolean value of true is returned, otherwise false is returned. If the right and left operands of the "not equal to" operator are not equal, the Boolean value of true is returned, otherwise false is returned. The following code has examples that compare all eight primitives to values of the same type.

```
int value = 12;
/* boolean comparison, prints true */
System.out.println(true == true);
/* char comparison, prints false */
System.out.println('a' != 'a');
/* byte comparison, prints true */
System.out.println((byte)value == (byte)value);
/* short comparison, prints false */
System.out.println((short)value != (short)value);
/* integer comparison, prints true */
System.out.println(value == value);
/* float comparison, prints true */
System.out.println(12F == 12f);
/* double comparison, prints false */
System.out.println(12D != 12d);
```

Reference values of objects can also be compared. Consider the following code:

```
Object a = new Object();
Object b = new Object();
Object c = b;
```

The reference variables are `a`, `b`, and `c`. As shown, reference variables `a` and `b` are unique. Reference variable `c` refers to reference variable `b`, so for equality purposes, they are the same.

```
/* Prints false, different references */
System.out.println(a == b);
/* Prints false, different references */
System.out.println(a == c);
/* Prints true, same references */
System.out.println(b == c);
```

The following are similar statements, but using the "not equal to" operator.

```
System.out.println(a != b); // Prints true, different references
System.out.println(a != c); // Prints true, different references
System.out.println(b != c); // Prints false, same references
```

**Numeric Promotion of Binary Values** By this point, you may be wondering what the compiler does with the operands when they are of different primitive types. Numeric promotion rules are applied on binary values for the additive (`+`, `-`), multiplicative (`*`, `/`, `%`), comparison (`<`, `<=`, `>`, `>=`), equality (`==`, `!=`), bitwise (`&`, `^`, `|`), and conditional (`?:`) operators. See Table 3-3.

## Logical Operators

Logical operators return Boolean values. Three are logical operators on the exam: logical AND, logical OR, and logical negation.

### Logical (Conditional) Operators

Logical (conditional) operators evaluate a pair of Boolean operands. Understanding their short-circuit principle is necessary for the exam.

- `&&` logical AND (conditional-AND) operator
- `||` logical OR (conditional-OR) operator

The logical AND operator evaluates the left and right operands. If both values of the operands have a value of true, then a value of true is returned. The logical AND is considered a short-circuit operator. If the left operand returns false, then there is no

**TABLE 3-3** Numeric Promotion of Binary Values

| Binary Numeric Promotion | |
|---|---|
| Check 1 | Check if one and only one operand is a `double` primitive. If so, convert the non-`double` primitive to a `double`, and stop checks. |
| Check 2 | Check if one and only one operand is a `float` primitive. If so, convert the non-`float` primitive to a `float`, and stop checks. |
| Check 3 | Check if one and only one operand is a `long` primitive. If so, convert the non-`long` primitive to a `long`, and stop checks. |
| Check 4 | Convert both operands to `int`. |

need to check the right operator since both would need to be true to return true; thus, it short-circuits. Therefore, whenever the left operand returns false, the expression terminates and returns a value of false.

```
/* Assigns true */
boolean and1 = true && true;
/* Assigns false */
boolean and2 = true && false;
/* Assigns false, right operand not evaluated */
boolean and3 = false && true;
/* Assigns false, right operand not evaluated */
boolean and4 = false && false;
```

The logical OR operator evaluates the left and right operands. If either value of the operands has a value of true, then a value of true is returned. The logical AND is considered a short-circuit operator. If the left operand returns true, there is no need to check the right operator, since either needs to be true to return true; thus, it short-circuits. Again, whenever the left operand returns true, the expression terminates and returns a value of true.

```
/* Assigns true, right operand not evaluated */
boolean or1 = true || true;
/* Assigns true, right operand not evaluated */
boolean or2 = true || false;
/* Assigns true */
boolean or3 = false || true;
/* Assigns false */
boolean or4 = false || false;
```

## Logical Negation Operator

The logical negation operator is also known as the inversion operator or Boolean invert operator. This is a simple operator, but don't take it lightly… you may see it quite often on the exam.

- `!` logical negation (inversion) operator

The logical negation operator returns the opposite of a Boolean value.

```
System.out.println(!false); // Prints true
System.out.println(!true); // Prints false
System.out.println(!!true); // Prints true
System.out.println(!!!true); // Prints false
System.out.println(!!!!true); // Prints true
```

Expect to see the logical negation operator used in conjunction with any method or expression that returns a Boolean value. The following list details some of these expressions that return Boolean values:

- Expressions with relational operators return Boolean values.
- Expressions with logical (conditional) operators return Boolean values.
- The `equals` method of the `Object` class returns Boolean values.
- The `String` methods `startsWith` and `endsWith` return Boolean values.

The following are some examples of statements that include the logical negation operator.

```
/* Example with relational expression */
iVar1 = 0;
iVar2 = 1;
if (!(iVar1 <= iVar2)) {};

/* Example with logical expressions */
boolean bVar1 = false; boolean bVar2 = true;
if ((bVar1 && bVar2) || (!(bVar1 && bVar2))){}

/* Example with equals method */
if (!"NAME".equals("NAME")) {}

/* Example with the String class's startsWith method */
String s = "Captain Jack";
System.out.println(!s.startsWith("Captain"));
```

The logical inversion operator cannot be used on a non-Boolean value. The following code will not compile:

```
!10; // compiler error, '!' must be used with a boolean value
not an integer
!"STRING"; // compiler error, '!' must be used with a boolean
value, not a string
```

Logical AND and logical OR are on the exam. Boolean AND and Boolean OR, along with bitwise AND and bitwise OR, are not on the exam. A reason why you may wish to use the nonlogical expressions associated with the right operand is if a change occurs to a variable where the new result is used later in your code. The following Scenario & Solution details the specifics of this scenario.

| SCENARIO & SOLUTION | |
|---|---|
| You wish to use an AND operator that evaluates the second operand whether the first operand evaluates to true or false. Which would you use? | Boolean AND (`&`) |
| You wish to use an OR operator that evaluates the second operand whether the first operand evaluates to true or false. Which would you use? | Boolean OR (`|`) |
| You wish to use an AND operator that evaluates the second operand only when the first operand evaluates to true. Which would you use? | Logical AND (`&&`) |
| You wish to use an OR operator that evaluates the second operand only when the first operand evaluates to false. Which would you use? | Logical OR (`||`) |

## CERTIFICATION OBJECTIVE

# Developing with String Objects and Their Methods

*Exam Objective 4.6 Develop code that uses the concatenation operator (+), and the following methods from class String: charAt, indexOf, trim, substring, replace, length, startsWith, and endsWith.*

Strings are commonly used in the Java programming language. This section discusses what strings are, how to concatenate separate strings, and then details the methods of the `String` class. When you have completed this section, which covers the following topics, you should fully understand what strings are and how to use them.

- Strings
- String concatenation operator
- Methods of the `String` class

## Strings

String objects are used to represent 16-bit Unicode character strings. Consider the 16 bits `000001011001` followed by `000001101111`. These bits in Unicode are represented as `\u0059` and `\u006F`. `\u0059` is mapped to the character "`Y`", and

`\u006F` is mapped to the character "`o`". An easy way of adding 16-bit Unicode character strings together in a reusable element is by declarating the data within a string.

```
String exclamation = "Yo"; // 000001011001 and 000001101111
```

See Appendix C for more information on the Unicode standard.

Strings are immutable objects, meaning their values never change. For example, the following text, "Dead Men Tell No Tales", can be created as a string.

```
String quote = "Dead Men Tell No Tales";
```

In the following example, the value of the string does not change after a `String` method returns a modified value. Remember that strings are immutable. Here, we invoke the `replace` method on the string. Again, the new string is returned, but will not change the value.

```
quote.replace("No Tales", "Tales"); // Returns new value
System.out.println(quote); // Prints the original value
$ Dead Men Tell No Tales
```

We can create strings in several ways. As with instantiating any object, you need to construct an object and assign it to a reference variable. As a reminder, a reference variable holds the value's address. Let's look at some of the things you can do with strings.

You can create a string without an assigned string object. Make sure you eventually give it a value, or you'll get a compiler error.

```
String quote1; // quote1 is a reference variable with no
assigned string object
quote1 = "Ahoy matey"; // Assigns string object to the reference
variable
```

You can use a couple of basic approaches to create a string object with an empty string representation.

```
String quote2a = new String(); // quote2a is a reference variable
String quote2b = new String(""); // Equivalent statement
```

You can create a string object without using a constructor.

```
String quote3 = "The existence of the sea means the existence
of Pirates! -- Malayan proverb";
```

| SCENARIO & SOLUTION | |
|---|---|
| You wish to use an object that represents an immutable character string. Which class will you use to create the object? | The `String` class |
| You wish to use an object that represents a thread-safe mutable character string. Which class will you use to create the object? | The `StringBuffer` class |
| You wish to use an object that represents a mutable character string. Which class will you use to create the object? | The `StringBuilder` class |

You can create a string object while using a constructor.

```
/* quote4 is a reference variable to the new string object */
String quote4 = new String("Yo ho ho!");
```

You can create a reference variable that refers to a separate reference variable of a string object.

```
String quote5 = "You're welcome to my gold -- William Kidd.";
String quote6 = quote5; // quote6 refers to the quote5 reference
variable of a string
```

You can assign a new string object to an existing string reference variable.

```
String quote7 = "The treasure is in the sand. "; // Assigns
string object to the reference variable
quote7 = "The treasure is between the rails."; // Assigns new
string to the same reference variable
```

If you wish to use a mutable character string, consider `StringBuffer` or `StringBuilder` as represented in the preceding Scenario & Solution.

## The String Concatenation Operator

The string concatenation operator concatenates (joins) strings together. The operator is denoted with the + sign.

- + String concatenation operator

If you have been programming for at least six months, odds are you have glued two strings together at some time. Java's string concatenation operator makes the act of joining two strings very easy. For example, "doub" + "loon" equates to "doubloon". Let's look at some more complete code.

```
String item = "doubloon";
String question = "What is a " + item + "? ";
System.out.println ("Question: " + question);
```

Line 2 replaces the `item` variable with its contents, "doubloon", and so the question string becomes:

```
What is a doubloon?
```

Notice that the question mark was appended as well.

Line 3 replaces the `question` variable with its contents and so the following string is returned:

```
$ Question: What is a doubloon?
```

It is that simple.

## The `toString` Method

The `Object` class has a method that returns a string representation of objects. This method is appropriately named the `toString` method. All classes in Java extend the `Object` class by default, so therefore every class inherits this method. When creating classes, it is common practice to override the `toString` method to return the data that best represents the state of the object. The `toString` method makes common use of the string concatenation operator.

Let's take a look at a `TreasureMap` class with the `toString` method overridden.

```
public class TreasureMap {
  private String owner = "Blackbeard";
  private String location = "Outer Banks";
  public String toString () {
    return "Map Owner: " + this.owner + ", treasure location: "
    + this.location;
  }
}
```

Here, the `toString` method returns the contents of the class's instance variables. Let's print out the representation of a `TreasureMap` object.

```
TreasureMap t = new TreasureMap();
System.out.println(t);
$ Map Owner: Blackbeard, treasure location: Outer Banks
```

Concatenation results may be unexpected if you are including variables that are not initially strings.

Consider a string and two integers:

```
String title1 = " shovels.";
String title2 = "Shovels: ";
int flatShovels = 5;
int roundPointShovels = 6;
```

The compiler performs left-to-right association for the additive and string concatenation operators. For the following two statements, the first two integers are added together. Next, the concatenation operator takes the `toString` representation of the result and concatenates it with the other string.

```
/* Prints '11 shovels' */
System.out.println(flatShovels + roundPointShovels + title1);

/* Prints '11 shovels' */
System.out.println((flatShovels + roundPointShovels) + title1);
```

Moving from left to right, the compiler takes the `title2` string and joins it with the string representation of the `flatShovels` integer variable. The result is a string. Now this result string is joined to the string representation of the `roundPointShovels` variable. Note that the `toString` method is used to return the string.

```
/* Prints 'Shovels: 56' */
System.out.println(title2 + flatShovels + roundPointShovels);
```

Parentheses take precedence, so you can join the sum of the integer values with the string if you code it as follows:

```
/* Prints 'Shovels: 11' */
System.out.println(title2 + (flatShovels + roundPointShovels));
```

## EXERCISE 3-2

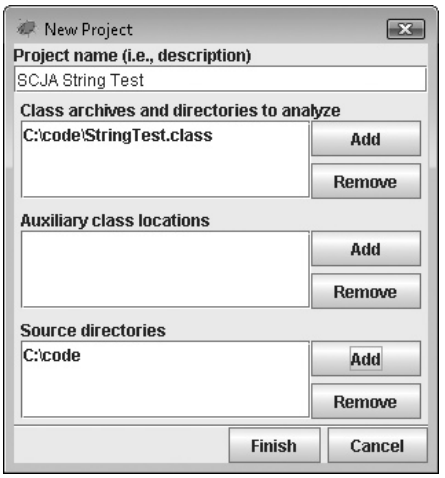
### Uncovering Bugs that Your Compiler May Not Find

Consider the strings in the following application:

```
public class StringTest {
  public static void main(String[] args) {
    String s1 = new String ("String one");
    String s2 = "String two";
    String s3 = "String " + "three";
  }
}
```

One of the strings is constructed in an inefficient manner. Do you know which one? Let's find out using the FindBugs application from the University of Maryland.

1. Create a directory named "`code`" somewhere on your PC (for example, `c:\code`).
2. Create the `StringTest.java` source file.
3. Compile the `StringTest.java` source file; javac StringTest.java.
4. Download the FindBugs software from `http://findbugs.sourceforge.net/`.
5. Extract, install, and run the FindBugs application. Note that the Eclipse and NetBeans IDEs have plug-ins for the FindBugs tool as well as other "software quality" tools such as PMD and Checkstyle.
6. Create a new project in FindBugs by choosing File and then New Project…
7. Add the project name (for instance, SCJA String Test).
8. Click the Add button for the text area associated with the "Class archives and directories to analyze." Find and select the `StringTest.class` file under the `C:\code` directory and click Choose.
9. Click the Add button for the text area associated with the "Source Directories." Find and select the `C:\code` directory (not the source file) and then click Choose.
10. The New Project dialog box will look similar to the following Illustration with the exception of your personal directory locations. Click Finish.




11. You will see that two bugs are returned. We are concerned with the first one. Drill down in the window that shows the bugs (Bugs | Performance | […]). The application will detail the warning and show you the source code with the line in error highlighted.
12. Fix the bug.

## Methods of the `String` Class

Several methods of the `String` class are commonly used. The common methods, `charAt`, `indexOf`, `length`, `replace`, `startsWith`, `endsWith`, `substring`, and `trim` were chosen by Sun to be included on the exam. These methods are detailed in Figure 3-2 and in the following section.

Coming up, we will be providing a description of each method, followed by the method declarations and associated examples.

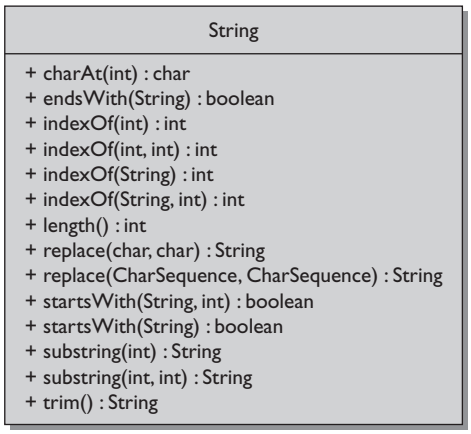
First, consider the following string:

```
String pirateMessage = "  Buried Treasure Chest! ";
```

The string has two leading blank spaces and one trailing blank space. This is important in relationship to the upcoming examples. The string is shown again in Figure 3-3 with the index values shown in relationship to the individual characters.

**FIGURE 3-2**

Commonly used methods of the `String` class




Let's use each method to perform some action with the string `pirateMessage`.

## The `charAt` Method

The `String` class's `charAt` method returns a primitive `char` value from a specified `int` index value in relationship to the referenced string object.

There is one `charAt` method declaration:

```
public char charAt(int index) {…}
```

Examples:

```
/* Returns the 'blank space' character from location 0 */
char c1 = pirateMessage.charAt(0);
/* Returns the character 'B' from location 2 */
char c2 = pirateMessage.charAt(2);
/* Returns the character '!' from location 23 */
char c3 = pirateMessage.charAt(23);
/* Returns the 'blank space' character from location 24 */
char c4 = pirateMessage.charAt(24);
```

**FIGURE 3-3** SCJA string detailing blanks

## The `indexOf` Method

The `String` class's `indexOf` methods return primitive `int` values representing the index of a character or string in relationship to the referenced string object.

Four `indexOf` method declarations exist:

```
public int indexOf(int ch) {…}
public int indexOf(int ch, int fromIndex) {…}
public int indexOf(String str) {…}
public int indexOf(String str, int fromIndex) {…}
```

Examples:

```
/* Returns the integer 3 as it is the first 'u' in the string. */
int i1 = pirateMessage.indexOf('u'); // 3
/* Returns the integer 14 as it is the first 'u' in the string
past location 9. */
int i2 = pirateMessage.indexOf('u', 9); // 14
/* Returns the integer 13 as it starts at location 13 in the
string. */
int i3 = pirateMessage.indexOf("sure"); // 13
/* Returns the integer -1 as there is no Treasure string on or
past location 10 */
int i4 = pirateMessage.indexOf("Treasure", 10); // -1!
/* Returns the integer -1 as there is no character u on or past
location 100 */
int i5 = pirateMessage.indexOf("u", 100); // -1!
```

## The `length` Method

The `String` class's `length` method returns a primitive `int` value representing the length of the referenced string object.

There is one `length` method declaration:

```
public int length() {…}
```

Examples:

```
/* Returns the string's length of 25 */
int i = pirateMessage.length(); // 25
```

**exam Watch**

***The*** `String` ***class uses the length method (for example,*** `string.length()`***). Arrays reference an instance variable in their state (for example,*** `array.length`***). Therefore, the string methods use the set of parentheses to return their length, and arrays do not. This is a gotcha that you will want to look for on the exam.***

```
// Use of String's length method
String string = "box";
string.length(); // 3
// Use of array's length attribute
String[] stringArray = new String[3];
stringArray.length; // 3
```

## The `replace` Method

The `String` class's `replace` method returns strings, replacing all characters or strings in relationship to the referenced string object. The `CharSequence` interface allows for the use of either a `String`, `StringBuffer`, or `StringBuilder` object.

Two `replace` method declarations can be used:

```
public String replace(char oldChar, char newChar) {…}
public String replace(CharSequence target, CharSequence
replacement) {…}
```

Examples:

```
/* Returns the string with all characters 'B' replaced with 'J'.
*/
String s1 = pirateMessage.replace('B', 'J'); // Juried Treasure
Chest!
/* Returns the string with all blank characters ' ' replaced
with 'X'. */
String s2 = pirateMessage.replace(' ', 'X'); // XXBuriedXTreasur
eXChest!X
/* Returns the string with all strings 'Chest' replaced with
'Coins'. */
String s3 = pirateMessage.replace("Chest", "Coins"); // Buried
Treasure Coins!
```

### The `startWith` Method

The `String` class's `startsWith` method returns a primitive `boolean` value representing the results of a test to see if the supplied prefix starts the referenced `String` object.

Two `startsWith` method declarations can be used:

```
public boolean startsWith(String prefix, int toffset) {…}
public boolean startsWith(String prefix) {…}
```

Examples:

```
/* Returns true as the referenced string starts with the
compared string. */
boolean b1 = pirateMessage.startsWith("  Buried Treasure"); //
true
/* Returns false as the referenced string does not start with
the compared string. */
boolean b2 = pirateMessage.startsWith("  Discovered"); // false
/* Returns false as the referenced string does not start with
the compared string at location 8. */
boolean b3 = pirateMessage.startsWith("Treasure", 8); // false
/* Returns true as the referenced string does start with the
compared string at location 9. */
boolean b4 = pirateMessage.startsWith("Treasure", 9); // true
```

### The `endsWith` Method

The `String` class's `endsWith` method returns a primitive `boolean` value representing the results of a test to see if the supplied suffix ends the referenced string object.

There is one `endsWith` method declaration:

```
public boolean endsWith(String suffix) {…}
```

Examples:

```
/* Returns true as the referenced string ends with the compared
string. */
boolean b1 = pirateMessage.endsWith("Treasure Chest! "); // true
/* Returns false as the referenced string does not end with the
compared string. */
boolean b2 = pirateMessage.endsWith("Treasure Chest "); // false
```

### The `substring` Method

The `String` class's `substring` methods return new strings that are substrings of the referenced string object.

Two `substring` method declarations exist:

```
public String substring(int beginIndex) {…}
public String substring(int beginIndex, int endIndex) {
```

Examples:

```
/* Returns the entire string starting at index 9. */
String s1 = pirateMessage.substring(9); // Treasure Chest!
/* Returns the string at index 9. */
String s2 = pirateMessage.substring(9, 10); // T
/* Returns the string at index 9 and ending at index 23. */
String s3 = pirateMessage.substring(9, 23); // Treasure Chest
/* Produces runtime error. */
String s4 = pirateMessage.substring(9, 8); // String index out
of range: -1
/* Returns a blank */
String s5 = pirateMessage.substring(9, 9); // Blank
```

### The `trim` Method

The `String` class's `trim` method returns the entire string minus leading and trailing whitespace characters in relationship to the referenced string object. The whitespace character corresponds to the Unicode character `\u0020`.

The sole `trim` method declaration is

```
public String trim() {…}
```

Examples:

```
/* "Buried Treasure Chest!" with no leading or trailing white
spaces */
String s = pirateMessage.trim();
```

on the job

***To view the source of the Java SE source files directly, either download the source code off the Internet or visit JDocs at `http://www.jdocs.com/`. JDocs provides an interface to the source code of several Java-based projects, including the Java platform, Standard Edition.***

# INSIDE THE EXAM

## Chaining

Java allows for methods to be chained together. Consider the following message from the captain of a pirate ship:

```
String msg = "  Maroon the First Mate with a flagon of water and a
pistol!  ";
```

We wish to change the message to read, "`Maroon the Quartermaster with a flagon of water`."

Three changes need to be made to adjust the string as desired:

1. Trim the leading and trailing whitespace.
2. Replace the substring "`First Mate`" with "`Quartermaster`".
3. Remove "`and a pistol`!"
4. Add a period at the end of the sentence.

A variety of methods and utilities can be used to make these changes. We will use the `trim`, `replace`, and `substring` methods, in this order.

```
msg = msg.trim(); // Trims whitespace
msg = msg.replace("First Mate", "Quartermaster");// Replaces text
msg = msg.substring(0,47);  // Returns first 48 characters.
```

Rather than writing these assignments individually, we can have one assignment statement with all of the methods chained. For simplicity, we also add the period with the string concatenation operator.

```
msg = msg.trim().replace("First Mate", "Quartermaster").substring(0,47) +
".";
```

Whether methods are invoked separately or chained together, the end result is the same.

```
System.out.println (msg);
$  Maroon the Quartermaster with a flagon of water.
```

Look for chaining on the exam.

## CERTIFICATION SUMMARY

This chapter discussed everything you need to know about operators and strings for the exam.

Operators of type assignment, arithmetic, relational, and logical were all presented in detail. Assignment operators included the general assignment, assignment by addition, and assignment by subtraction operators. Arithmetic operators included the addition, subtraction, multiplication, division, and remainder (modulus) operators, as well as the prefix increment, prefix decrement, postfix increment, and postfix decrement operators. Relational operators included the "less than," "less than or equal to," "greater than," "greater than or equal to," "equal to," and "not equal to" operators. Logical operators included the logical negation, logical AND, and logical OR operators.

Strings were discussed in three main areas: creating strings, the string concatenation operator, and methods of the `String` class. The following methods of the `String` class were covered: `charAt`, `indexOf`, `length`, `replace`, `startsWith`, `endsWith`, `substring`, and `trim`.

Knowing the fine details of these core areas related to operators and strings is necessary for the exam.

# TWO-MINUTE DRILL

## Understanding Fundamental Operators

- ❑ The exam covers the following assignment and compound assignment operators: `=`, `+=`, and `-=`.
- ❑ The assignment operator (`=`) assigns values to variables.
- ❑ The additional compound assignment operator is used for shorthand. As such, `a=a+b` is written `a+=b`.
- ❑ The subtraction compound assignment operator is used for shorthand. As such, `a=a-b` is written `a-=b`.
- ❑ The exam covers the following arithmetic operators: `+`, `-`, `*`, `/`, `%`, `++`, and `--`.
- ❑ The addition operation (`+`) is used to add two operands together.
- ❑ The subtraction operator (`-`) is used to subtract the right operand from the left operand.
- ❑ The multiplication operator (`*`) is used to multiply two operands together.
- ❑ The divisor operator (`/`) is used to divide the right operand into the left operand.
- ❑ The modulus operator (`%`) returns the remainder of a division.
- ❑ The prefix increment (`++`) and prefix decrement (`--`) operators are used to increment or decrement a value before it is used in an expression.
- ❑ The postfix increment (`++`) and postfix decrement (`--`) operators are used to increment or decrement a value after it is used in an expression.
- ❑ The exam covers the following relational operators: `<`, `<=`, `>`, `>=`, `==`, and `!=`.
- ❑ The "less than" operator (`<`) returns `true` if the left operand is less than the right operand.
- ❑ The "less than or equal to" operator (`<=`) returns `true` if the left operand is less than or equal to the right operand.
- ❑ The "greater than" operator (`>`) returns `true` if the right operand is less than the left operand.
- ❑ The "greater than or equal to" operator (`>=`) returns `true` if the right operand is less than or equal to the left operand.

- ❑ The "equal to" equality operator (==) returns `true` if the left operand is equal to the right operand.
- ❑ The "not equal to" equality operator (`!=`) returns `true` if the left operand is not equal to the right operand.
- ❑ Equality operators can test numbers, characters, Booleans, and reference variables.
- ❑ The exam covers the following logical operators: `!`, `&&`, and `||`.
- ❑ The logical negation (inversion) operator (`!`) negates the value of the `boolean` operand.
- ❑ The logical AND (conditional AND) operator (`&&`) returns `true` if both operands are `true`.
- ❑ The logical AND operator is known as a short-circuit operator because it does not evaluate the right operand if the left operand is `false`.
- ❑ The logical OR (conditional OR) operator (`||`) returns `true` if either operand is `true`.
- ❑ The conditional OR operator is known as a short-circuit operator because it does not evaluate the right operand if the left operand is `true`.

## Developing with String Objects and Their Methods

- ❑ An object of the `String` class represents an immutable character string.
- ❑ An object of the `StringBuilder` class represents a mutable character string.
- ❑ An object of the `StringBuffer` class represents a thread-safe mutable character string.
- ❑ Mutable means "changeable." Note that Java variables such as primitives are mutable by default and can be made immutable by using the `final` keyword.
- ❑ The `CharSequence` interface is implemented by the `String`, `StringBuilder`, and `StringBuffer` classes. It can be used as an argument in the `String` class's `replace` method.
- ❑ The string concatenation operator (`+`) joins two strings together.
- ❑ The string concatenation operator will join two operands together, as long as one or both of them are strings.
- ❑ The `String` class's `charAt` method returns a primitive `char` value from a specified `int` index value in relationship to the referenced string.

- The `String` class's `indexOf` methods returns a primitive `int` value representing the index of a character or string in relationship to the referenced string.
- The `String` class's `length` method returns a primitive `int` value representing the length of the referenced string.
- The `String` class's `replace` methods return strings replacing all characters or strings in relationship to the referenced string.
- The `String` class's `startsWith` method returns a primitive `boolean` value representing the results of a test to see if the supplied prefix starts the referenced string.
- The `String` class's `endsWith` method returns a primitive `boolean` value representing the results of a test to see if the supplied suffix ends the referenced string.
- The `String` class's `substring` methods return new strings that are substrings of the referenced string.
- The `String` class's `trim` method returns the entire string minus leading and trailing whitespace characters in relationship to the referenced string.

# SELF TEST

## Understanding Fundamental Operators

**1.** Given:

```
public class ArithmeticResultsOutput {
  public static void main (String[] args) {
    int i = 0;
    int j = 0;
    if (i++ == ++j) {
      System.out.println("True: i=" + i + ", j=" + j);
    } else {
      System.out.println("False: i=" + i + ", j=" + j);
    }
  }
}
```

What will be printed to standard out?

A. `True: i=0, j=1`

B. `True: i=1, j=1`

C. `False: i=0, j=1`

D. `False: i=1, j=1`

**2.** Which set of operators represents the complete set of valid Java assignment operators?

A. `%=, &=, *=, $=, :=, /=, ^=, |=, +=, <<=, =, -=, >>=, >>>=`

B. `%=, &=, *=, /=, ^=, |=, +=, <<=, <<<=, =, -=, >>=, >>>=`

C. `%=, &=, *=, /=, ^=, |=, +=, <<=, =, -=, >>=, >>>=`

D. `%=, &=, *=, $=, /=, ^=, |=, +=, <<=, <<<=, =, -=, >>=, >>>=`

**3.** Given the following Java code segment, what will be printed, considering the usage of the modulus operators?

```
System.out.print(49 % 26 % 5 % 1);
```

A. 23

B. 3

C. 1

D. 0

**4.** Given:

```
public class BooleanResultsOutput {
  public static void main (String[] args) {
    boolean booleanValue1 = true;
    boolean booleanValue2 = false;
    System.out.print(!(booleanValue1 & !booleanValue2) + ", ");
    System.out.print(!(booleanValue1 | !booleanValue2)+ ", ");
    System.out.print(!(booleanValue1 ^ !booleanValue2));
  }
}
```

What will be printed, considering the usage of the logical Boolean operators?

A. false, false, true

B. false, true, true

C. true, false, true

D. true, true, true

**5.** Given:

```
public class ArithmeticResultsOutput {
  public static void main (String[] args) {
    int i1 = 100; int j1 = 200;
    if ((i1 == 99) & (--j1 == 199)) {
      System.out.print("Value1: " + (i1 + j1) + " ");
    } else {
      System.out.print("Value2: " + (i1 + j1) + " ");
    }
    int i2 = 100; int j2 = 200;
    if ((i2 == 99) && (--j2 == 199)) {
      System.out.print("Value1: " + (i2 + j2) + " ");
    } else {
      System.out.print("Value2: " + (i2 + j2) + " ");
    }
    int i3 = 100; int j3 = 200;
    if ((i3 == 100) | (--j3 == 200)) {
      System.out.print("Value1: " + (i3 + j3) + " ");
    } else {
      System.out.print("Value2: " + (i3 + j3) + " ");
    }
    int i4 = 100; int j4 = 200;
    if ((i4 == 100) || (--j4 == 200)) {
      System.out.print("Value1: " + (i4 + j4) + " ");
```

```
      } else {
       System.out.print("Value2: " + (i4 + j4) + " ");
      }
    }
  }
```

What will be printed to standard out?

A. `Value2: 300 Value2: 300 Value1: 300 Value1: 300`

B. `Value2: 299 Value2: 300 Value1: 299 Value1: 300`

C. `Value1: 299 Value1: 300 Value2: 299 Value2: 300`

D. `Value1: 300 Value1: 299 Value2: 300 Value2: 299`

**6.** Given the following code segment:

```
public void validatePrime() {
  long p = 17496; // 'prime number' candidate
  Double primeSquareRoot = Math.sqrt(p);
  boolean isPrime = true;
  for (long j = 2; j <= primeSquareRoot.longValue(); j++) {
    if (p % j == 0) {
      // Print divisors
      System.out.println(j + "x" + p / j);
      isPrime = false;
    }
  }
  System.out.println("Prime number: " + isPrime);
}
```

Which of the following is true? Hint: 17496 is not a prime number.

A. The code will not compile due to a syntactical error somewhere in the code.

B. The code will not compile since the expression `(p % j == 0)` should be written as `((p % j) == 0)`.

C. Divisors will be printed to standard out (for example, `2x8478`, and so on), along with "`Prime number: false`" as the final output.

D. Divisors will be printed to standard out (for example, `2x8478`, and so on), along with "`Prime number: 0`" as the final output.

**7.** Given:

```
public class EqualityTests {
  public static void main (String[] args) {
    Integer value1 = new Integer("312");
    Integer value2 = new Integer("312");
    Object object1 = new Object();
    Object object2 = new Object();
    Object object3 = value1;
  }
}
```

Which expressions evaluate to `true`?

A. `value1.equals(value2)`

B. `value1.equals(object1)`

C. `value1.equals(object3)`

D. `object1.equals(object2)`

## Developing with String Objects and Their Methods

**8.** Given:

```
System.out.print(3 + 3 + "3");
System.out.print(" and ");
System.out.println("3" + 3 + 3);
```

What will be printed to standard out?

A. `333 and 333`

B. `63 and 63`

C. `333 and 63`

D. `63 and 333`

**9.** Consider the interface `CharSequence` that is a required argument in one of the `replace` method declarations:

```
public String replace(CharSequence target, CharSequence
replacement) {
   …
}
```

This `CharSequence` interface is a super interface to which concrete classes?

A. `String`

B. `StringBoxer`

C. `StringBuffer`

D. `StringBuilder`

**10.** Which statement is false about the `toString` method?

A. The `toString` method is a method of the `Object` class.

B. The `toString` method returns a string representation of the object.

C. The `toString` method must return the object's state information in the form of a string.

D. The `toString` method is commonly overridden.

**11.** Which `indexOf` method declaration is invalid?

A. `indexOf(int ch)`

B. `indexOf(int ch, int fromIndex)`

C. `indexOf(String str, int fromIndex)`

D. `indexOf(CharSequence str, int fromIndex)`

**12.** Given:

```
String tenCharString = "AAAAAAAAAA";
System.out.println(tenCharString.replace("AAA", "LLL"));
```

What is printed to the standard out?

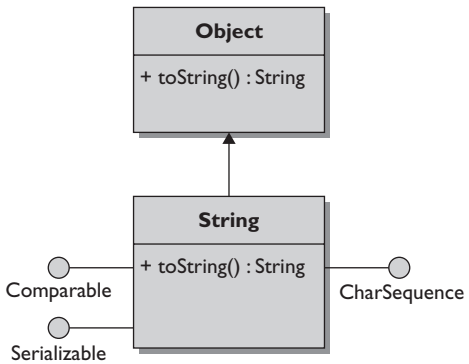
A. `AAAAAAAAAA`

B. `LLLAAAAAAA`

C. `LLLLLLLLLA`

D. `LLLLLLLLL`

**13.** Consider the following illustration. Which statements, also represented in the illustration, are true?




A. The `String` class implements the `Object` interface.

B. The`String` class implements the `Comparable`, `Serializable`, and `CharSequence` interfaces.

C. The `toString` method overrides the `toString` method of the `Object` class, allowing the `String` object to return its own string.

D. The `toString` method is publicly accessible.

# SELF TEST ANSWERS

## Understanding Fundamental Operators

**1.** Given:

```
public class ArithmeticResultsOutput {
  public static void main (String[] args) {
    int i = 0;
    int j = 0;
    if (i++ == ++j) {
      System.out.println("True: i=" + i + ", j=" + j);
    } else {
      System.out.println("False: i=" + i + ", j=" + j);
    }
  }
}
```

What will be printed to standard out?

A. `True: i=0, j=1`

B. `True: i=1, j=1`

C. `False: i=0, j=1`

D. `False: i=1, j=1`

Answer:

☑ **D.** The value of `j` is prefix incremented before the evaluation; however, the value of `i` is not. Therefore, the expression is evaluated with a `boolean` value of `false` as a result since 0 does not equal 1 (that is, i=0 and j=1). After the expression has been evaluated, but before the associated print statement is executed, the value of `i` is postfix incremented (that is, (i=1)). Therefore, the correct answer is False: i=1, j=1).

☒ **A, B,** and **C** are incorrect answers as justified by the correct answer's explanation.

**2.** Which set of operators represents the complete set of valid Java assignment operators?

A. `%=, &=, *=, $=, :=, /=, ^=, |=, +=, <<=, =, -=, >>=, >>>=`

B. `%=, &=, *=, /=, ^=, |=, +=, <<=, <<<=, =, -=, >>=, >>>=`

C. `%=, &=, *=, /=, ^=, |=, +=, <<=, =, -=, >>=, >>>=`

D. `%=, &=, *=, $=, /=, ^=, |=, +=, <<=, <<<=, =, -=, >>=, >>>=`

Answer:

☑ **C.** The complete set of valid Java assignment operators is represented.

☒ **A, B,** and **D** are incorrect answers. **A** is incorrect since `$=` and `:=` are not valid Java assignment operators. **B** is incorrect because `<<<=` is not a valid Java assignment operator. **D** is incorrect because `$=` and `<<<=` are not valid Java assignment operators.

**3.** Given the following Java code segment, what will be printed, considering the usage of the modulus operators?

```
System.out.print(49 % 26 % 5 % 1);
```

A. 23

B. 3

C. 1

D. 0

Answer:

☑ **D.** The remainder of 49/26 is 23. The remainder of 23/5 is 3. The remainder of 3/1 is 0. The answer is 0.

☒ **A, B,** and **C** are incorrect answers as justified by the correct answer's explanation.

**4.** Given:

```
public class BooleanResultsOutput {
  public static void main (String[] args) {
    boolean booleanValue1 = true;
    boolean booleanValue2 = false;
    System.out.print(!(booleanValue1 & !booleanValue2) + ", ");
    System.out.print(!(booleanValue1 | !booleanValue2)+ ", ");
    System.out.print(!(booleanValue1 ^ !booleanValue2));
  }
}
```

What will be printed, considering the usage of the logical Boolean operators?

A. false, false, true

B. false, true, true

C. true, false, true

D. true, true, true

Answer:

☑ **A.** The first expression statement `(!(true & !(false)))` evaluates to `false`. Here, the right operand is negated to `true` by the (Boolean invert) operator, the Boolean AND operator equates the expression of the two operands to `true`, and the (Boolean invert) operator equates the resultant value to `false`. The second expression statement `(!(true | !(false)))` evaluates to `false`. Here, the right operand is negated to `true` by the (Boolean invert) operator, the Boolean OR operator equates the expression of the two operands to `true`, and the (Boolean invert) operator equates the resultant value to `false`. The third expression statement `(!(true ^ !(false)))` evaluates to `true`. Here, the right operand is negated to `true` by the (Boolean invert) operator, the Boolean XOR operator equates the expression of the two operands to `false`, and the (Boolean invert) operator equates the resultant value to `true`.

☒ **B, C,** and **D** are incorrect answers as justified by the correct answer's explanation.

**5.** Given:

```
public class ArithmeticResultsOutput {
  public static void main (String[] args) {
    int i1 = 100; int j1 = 200;
    if ((i1 == 99) & (--j1 == 199)) {
      System.out.print("Value1: " + (i1 + j1) + " ");
    } else {
      System.out.print("Value2: " + (i1 + j1) + " ");
    }
    int i2 = 100; int j2 = 200;
    if ((i2 == 99) && (--j2 == 199)) {
      System.out.print("Value1: " + (i2 + j2) + " ");
    } else {
      System.out.print("Value2: " + (i2 + j2) + " ");
    }
    int i3 = 100; int j3 = 200;
    if ((i3 == 100) | (--j3 == 200)) {
      System.out.print("Value1: " + (i3 + j3) + " ");
    } else {
      System.out.print("Value2: " + (i3 + j3) + " ");
    }
    int i4 = 100; int j4 = 200;
    if ((i4 == 100) || (--j4 == 200)) {
      System.out.print("Value1: " + (i4 + j4) + " ");
     } else {
      System.out.print("Value2: " + (i4 + j4) + " ");
    }
  }
}
```

What will be printed to standard out?

A. `Value2: 300 Value2: 300 Value1: 300 Value1: 300`

B. `Value2: 299 Value2: 300 Value1: 299 Value1: 300`

C. `Value1: 299 Value1: 300 Value2: 299 Value2: 300`

D. `Value1: 300 Value1: 299 Value2: 300 Value2: 299`

Answer:

☑ **B** is the correct because `Value2: 299 Value2: 300 Value1: 299 Value1: 300` will be printed to the standard out. Note that `&&` and `||` are short-circuit operators. So… When the first operand of a conditional AND (`&&`) expression evaluates to `false`, the second operand is not evaluated. When the first operand of a conditional OR (`||`) expression evaluates to `true`, the second operand is not evaluated. Thus, for the second and fourth `if` statements, the second operand isn't evaluated. Therefore, the prefix increment operators are never executed and do not affect the values of the j[x] variables.

☒ **A, C,** and **D** are incorrect answers as justified by the correct answer's explanation.

**6.** Given the following code segment:

```
public void validatePrime() {
  long p = 17496; // 'prime number' candidate
  Double primeSquareRoot = Math.sqrt(p);
  boolean isPrime = true;
  for (long j = 2; j <= primeSquareRoot.longValue(); j++) {
    if (p % j == 0) {
      // Print divisors
      System.out.println(j + "x" + p / j);
      isPrime = false;
    }
  }
  System.out.println("Prime number: " + isPrime);
}
```

Which of the following is true? Hint: 17496 is not a prime number.

A. The code will not compile due to a syntactical error somewhere in the code.

B. The code will not compile since the expression `(p % j == 0)` should be written as `((p % j) == 0)`.

C. Divisors will be printed to standard out (for example,`2x8478`, and so on), along with "`Prime number: false`" as the final output.

D. Divisors will be printed to standard out (for example,`2x8478`, and so on), along with "`Prime number: 0`" as the final output.

Answer:

☑ **C.** Divisors will be printed to standard out followed by "`Prime number: false`". For those curious, the complete list of divisors printed are `2x8748`, `3x5832`, `4x4374`, `6x2916`, `8x2187`, `9x1944`, `12x1458`, `18x972`, `24x729`, `27x648`, `36x486`, `54x324`, `72x243`, `81x216`, and `108x162`.

☒ **A, B,** and **D** are incorrect answers. **A** is incorrect because there are no syntactical errors in the code. **B** is incorrect because a set of parentheses around "`p % j`" is not required. Answer **D** is incorrect because the code does not print out the character `0`, it prints out the `boolean` literal value `false`.

**7.** Given:

```
public class EqualityTests {
  public static void main (String[] args) {
    Integer value1 = new Integer("312");
    Integer value2 = new Integer("312");
    Object object1 = new Object();
    Object object2 = new Object();
    Object object3 = value1;
  }
}
```

Which expressions evaluate to `true`?

A. `value1.equals(value2)`

B. `value1.equals(object1)`

C. `value1.equals(object3)`

D. `object1.equals(object2)`

Answer:

☑ **A** and **C. A** is correct because the class Integer implements the Comparable interface, allowing use of the `equals` method. **C** is correct because the `Integer` object was used to create the `Object` reference.

☒ **B** and **D** are incorrect because the code cannot equate two objects with different references.

## Developing with String Objects and Their Methods

**8.** Given:

```
System.out.print(3 + 3 + "3");
System.out.print(" and ");
System.out.println("3" + 3 + 3);
```

What will be printed to standard out?

A. `333 and 333`

B. `63 and 63`

C. `333 and 63`

D. `63 and 333`

Answer:

☑ **D.** The (+) operators have left-to-right association. The first two operands of the first statement are numeric, so the addition (+) operator is used. Therefore, `3 + 3 = 6`. Since `6 + "3"` uses a string as an operand, the string concatenation (+) operator is used. Therefore, concatenating the strings "`6`" and "`3`" renders the string "`63`". The last statement is handled a little differently. The first operand is a `String`, therefore the string concatenation operator is used with the other operands. Thus, concatenating strings "3" + "3" + "3" renders the string "333". The correct answer is "63 and 333".

☒ **A, B,** and **C** incorrect. Note that changing `("3" + 3 + 3)` to `("3" + (3 + 3))` would have rendered "`36`".

**9.** Consider the interface `CharSequence` that is a required argument in one of the `replace` method declarations:

```
public String replace(CharSequence target, CharSequence replacement) {
    …
}
```

This `CharSequence` interface is a super interface to which concrete classes?

A. `String`

B. `StringBoxer`

C. `StringBuffer`

D. `StringBuilder`

Answer:

☑ **A, C,** and **D.** The concrete classes `String`, `StringBuffer`, and `StringBuilder` all implement the interface `CharSequence`. These classes can all be used in a polymorphic manner in regards to CharSequence being an expected argument in one of the `String` class's `replace` methods.

☒ **B** is incorrect. There is no such thing as a `StringBoxer` class.

**10.** Which statement is false about the `toString` method?

A. The `toString` method is a method of the `Object` class.

B. The `toString` method returns a string representation of the object.

C. The `toString` method must return the object's state information in the form of a string.

D. The `toString` method is commonly overridden.

Answer:

☑ **C.** While the `toString` method is commonly used to return the object's state information, any information that can be gathered may be returned in the string.

☒ **A, B,** and **D** are incorrect answers since they all represent true statements. **A** is incorrect because the `toString` method is a method of the `Object` class. **B** is incorrect because the `toString` method returns a string representation of the object. **D** is incorrect because the `toString` method is also commonly overridden.

**11.** Which `indexOf` method declaration is invalid?

A. `indexOf(int ch)`

B. `indexOf(int ch, int fromIndex)`

C. `indexOf(String str, int fromIndex)`

D. `indexOf(CharSequence str, int fromIndex)`

Answer:

☑ **D.** The method declaration including `indexOf(CharSequence str, int fromIndex)` is invalid. `CharSequence` is not used as an argument in any `indexOf` method. Note that `String`, `StringBuffer`, and `StringBuilder` all declare their own `indexOf` methods.

☒ **A, B,** and **C** are incorrect because they are all valid method declarations.

**12.** Given:

```
String tenCharString = "AAAAAAAAAA";
System.out.println(tenCharString.replace("AAA", "LLL"));
```

What is printed to the standard out?

A. AAAAAAAAAA

B. LLLAAAAAAA

C. LLLLLLLLLA

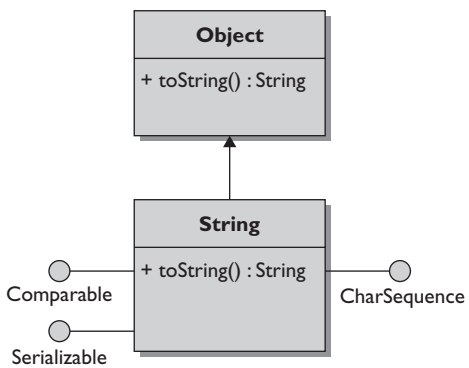
D. LLLLLLLLL

Answer:

☑ **C.** The `replace` method of the `String` class replaces all instances of the specified string. The first three instances of AAA are replaced by `LLL`, making `LLLLLLLLLA`.

☒ **A, B,** and **D** are incorrect answers as justified by the correct answer's explanation.

**13.** Consider the following illustration. Which statements, also represented in the illustration, are true?




A. The `String` class implements the `Object` interface.

B. The `String` class implements the `Comparable`, `Serializable`, and `CharSequence` interfaces.

C. The `toString` method overrides the `toString` method of the `Object` class, allowing the `String` object to return its own string.

D. The `toString` method is publicly accessible.

Answer:

☑ **B, C,** and **D** are all correct because they represent true statements. **B** is correct because the `String` class implements the `Comparable`, `Serializable`, and `CharSequence` interfaces. **C** is correct because the `toString` method overrides the `toString` method of the `Object` class, allowing the `String` object to return its own string. **D** is correct because the `toString` method is also publicly accessible.

☒ **A** is incorrect. The `Object` class is a concrete class. Therefore, the `String` class does not implement an `Object` interface since there is no such thing as an `Object` interface. The `String` class actually extends an `Object` concrete class.